

ANSI C Toolset Reference Manual

INMOS Limited



INMOS is a member of the SGS-THOMSON Microelectronics Group

© INMOS Limited 1992. This document may not be copied, in whole or in part, without prior written consent of INMOS.

[®], **inmos**[®], IMS and occam are trademarks of INMOS Limited.

INMOS Limited is a member of the SGS-THOMSON Microelectronics Group.



is a registered trademark of the SGS-THOMSON Microelectronics Group.

The C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler.

INMOS Document Number: 72 TDS 346 01

Contents overview

Contents

Preface

Tools

1	<i>icc</i>	Describes the ANSI C compiler.
2	<i>icconf</i>	Describes the configurer which generates configuration binary files from configuration descriptions.
3	<i>icollect</i>	Describes the code collector which generates executable code files.
4	<i>idebug</i>	Describes the network debugger. Lists the symbolic functions and Monitor page commands at machine level.
5	<i>idump</i>	Describes the memory dumper tool which dumps root transputer memory for post mortem debugging.
6	<i>iemit</i>	Describes the memory configurer tool which helps to configure the transputer memory interface.
7	<i>ieprom</i>	Describes the EPROM formatter tool which creates executable files for loading into ROM.
8	<i>ilibr</i>	Describes the toolset librarian which creates libraries from compiled code files.
9	<i>ilink</i>	Describes the toolset linker which links compiled code and libraries into a single unit.
10	<i>ilist</i>	Describes the binary lister which displays binary files in a readable form.
11	<i>imakef</i>	Describes the Makefile generator which creates Makefiles for toolset compilations.
12	<i>imap</i>	Describes the map tool which generates a memory map for an executable file.
13	<i>iserver</i>	Describes the host file server which loads programs onto transputer hardware and provides host communication.
14	<i>isim</i>	Describes the transputer simulator which allows programs to be run without hardware.
15	<i>iskip</i>	Describes the skip loader tool which loads programs onto external subnetworks.

Appendices

A	<i>Toolset standards and conventions</i>	Describes the conventions and standards of the toolset.
B	<i>Transputer types and classes</i>	Describes the meaning and use of transputer types and classes and lists the command line options to select them for the compiler and linker.
C	<i>Using the assembler</i>	Describes the use of the C assembler and the assembler directives.
D	<i>ISERVER Protocol</i>	Describes the server protocol and the ISERVER functions.
E	<i>ITERM</i>	Describes the format of the ITERM files.
F	<i>Bootstrap loaders</i>	Describes the INMOS bootstrap loading scheme and advises on how it might be customized.

Index

Contents

Contents overview	i
Contents	iii
Preface	xix
Host versions	xix
About this manual	xix
About the toolset documentation set	xx
Other documents	xxi
occam and FORTRAN toolsets	xxi
Documentation conventions	xxi
Tools	1
1 icc — ANSI C compiler	3
1.1 Introduction	3
1.2 Running the compiler	3
Examples of use:	6
1.2.1 Optimizing compiler options	6
1.2.2 Transputer targets	6
1.2.3 Error modes	7
1.2.4 Default command line options	7
1.2.5 File extension defaults	7
1.2.6 Search paths	7
1.2.7 Using the assembler	7
1.2.8 Compatibility with other C implementations	8
Arithmetic right shifts	8
Signedness of char	8
1.2.9 Software quality check	8
1.3 Memory map	9
1.4 Compiler directives	12
1.4.1 #define	12
1.4.2 #elif constant_expression	12
1.4.3 #else	12
1.4.4 #endif	12
1.4.5 #error	13
1.4.6 #if	13
1.4.7 #ifdef	13
1.4.8 #ifndef	13

1.4.9	#include	14
	Relative directory names	14
	Backslash character in filenames	14
1.4.10	#line	14
1.4.11	#pragma	14
	Pragma IMS_nolink	17
	Pragma IMS_descriptor	17
1.4.12	#undef	19
1.5	Compiler predefinitions	19
1.5.1	Macro names	19
1.5.2	Other predefines	20
1.6	Transputer inline code	21
1.6.1	Inlined functions	21
1.7	Compiler diagnostics	22
1.7.1	Message format	22
1.7.2	Severities	22
1.7.3	Standard terms	22
	abstract declarator	23
1.7.4	ANSI trigraphs	24
1.7.5	Warning diagnostics	24
1.7.6	Recoverable errors	31
1.7.7	Serious errors	38
2	icconf - configurer	49
2.1	Introduction	49
2.2	Configuration language implementation	50
2.3	Running the configurer	50
2.3.1	Default command line	52
2.3.2	Virtual routing processes	52
2.3.3	Support for the Advanced Toolset	52
2.3.4	Boot from ROM options	52
2.3.5	Mixed language programming	53
2.3.6	Configurer library file	53
2.3.7	Standard include files	53
	Defaults file setconf.inc	53
	Other include files	53
2.3.8	Configuration description examples	53
2.3.9	Search paths	54
2.3.10	Default memory map	54
2.3.11	LoadStart	55
2.3.12	System processes	55
2.4	Configurer messages	55
2.4.1	Information	56
2.4.2	Warnings	57

2.4.3	Errors	60
2.4.4	Serious messages	75
2.4.5	Fatal errors	79
3	icollect — code collector	81
3.1	Introduction	81
	Unconfigured program (using 'T' option):	82
	Configured processor program:	82
3.2	Running the code collector	82
3.2.1	Examples of use	85
	Example A (unconfigured program mode):	85
3.2.2	Default command line	85
3.2.3	Input files	85
3.2.4	Output files	85
	Single processor non-configured case (T option) ..	86
	Configured programs	86
	Memory map files	86
	Debug data file	86
3.3	Memory allocation for unconfigured programs	86
3.3.1	C and FORTRAN programs	87
3.3.2	occam programs	88
3.3.3	Memory initialization errors	89
3.3.4	Small values of IBOARDSize	89
3.4	Parity-checked memory	90
3.5	Non-bootable files created with the K option	91
3.5.1	File format	91
3.6	Boot-from-ROM output files	92
3.7	Alternative bootstrap loaders for unconfigured programs	93
3.8	Alternative bootstrap schemes	93
3.9	The memory map file	93
3.9.1	Unconfigured (single processor), boot from link ...	94
	Program targetted at transputer type	94
	Program targetted at transputer class	96
3.9.2	Configured program boot from link	97
3.9.3	Boot from ROM programs	98
	Unconfigured (single processor), boot from ROM,	
	run in RAM	98
	Unconfigured (single processor), boot from ROM,	
	run in ROM	98
	Configured program, boot from ROM, run in RAM .	98
	Configured program, boot from ROM, run in ROM .	98
3.10	Disabling interactive debugging – 'Y' option	99
3.11	Error messages	100
3.11.1	Warnings	100

3.11.2	Serious errors	100
3.11.3	Fatal errors	106
4	idebug — network debugger	107
4.1	Introduction	107
4.2	Debugging the root transputer	107
4.2.1	Board wiring	108
4.2.2	Post-mortem debugging R-mode programs	108
4.2.3	Post-mortem debugging T-mode programs	108
4.2.4	Post-mortem debugging from a network dump file	109
4.2.5	Debugging a dummy network	109
4.2.6	Methods for interactive breakpoint debugging	109
4.3	Running the debugger	109
4.3.1	Toolset file types read by the debugger	111
4.3.2	Environment variables	112
4.3.3	Program termination	113
4.4	Post-mortem mode invocation	113
4.4.1	Debugging T-mode programs — option 'T'	114
4.4.2	Debugging R-mode programs — option 'R'	114
4.4.3	Debugging a network dump file — option 'N'	114
4.4.4	Debugging a previous breakpoint session — option 'M'	115
4.4.5	Reinvoking the debugger on single transputer programs	115
4.4.6	Debugging boot from ROM programs	115
4.5	Interactive mode invocation	115
4.6	Function key mappings	116
4.7	Debugging programs on INMOS boards	116
4.7.1	Subsystem wiring	116
4.7.2	Debugging options to use with specific board types	117
4.7.3	Detecting the error flag in interactive mode	117
4.8	Debugging programs on non-INMOS boards	118
4.9	Monitor page commands	119
4.9.1	Command format	119
4.9.2	Specifying transputer addresses	119
4.9.3	Scrolling the display	119
4.9.4	Editing functions	120
4.9.5	Commands mapped by ITERM	120
4.9.6	Summary of commands	120
4.9.7	Symbolic-type commands	122
4.9.8	Scroll keys	122
4.9.9	Monitor page command descriptions	123
4.9.10	Symbolic-type commands	141
4.10	Symbolic functions	142

4.10.1	Symbolic functions	143
4.10.2	Interactive mode functions	144
4.10.3	Locating functions	146
4.10.4	Cursor and display control functions	146
4.10.5	Miscellaneous functions	147
4.11	INSPECT/MODIFY expression language for C	149
4.11.1	Syntax not supported	149
4.11.2	Extensions to C syntax	149
	Subarrays	149
	Scope resolution operator	149
	Hex constants	150
	Address constant indirect	150
4.11.3	Automatic expression pickup	150
4.11.4	Editing functions	151
4.11.5	Warnings	151
4.11.6	Types	152
	Type compatibility when using	152
4.12	Display formats for source code symbols	153
4.12.1	Notation	153
4.12.2	Basic Types	154
4.12.3	Default type of "plain" char	154
4.12.4	Enumerated types	154
4.12.5	Pointers	155
4.12.6	Function Pointers	155
4.12.7	Structs	155
4.12.8	Unions	156
4.12.9	Addressof operator &	156
4.12.10	Arrays	156
4.12.11	Channels	157
4.13	Example displays	157
4.14	INSPECT/MODIFY expression language for occam	159
4.14.1	Inspecting memory	159
4.14.2	Inspecting arrays	159
4.14.3	Type compatibility when using	160
4.15	Display formats for source code symbols	161
4.15.1	Notation	161
4.15.2	Basic Types	161
4.15.3	Channels	162
4.15.4	Arrays	163
4.15.5	Procedures and functions	163
4.16	Example displays	163
4.17	Error messages	166
4.17.1	Out of memory errors	166
4.17.2	If the debugger hangs	166
4.17.3	Error message list	166

5	idump — memory dumper	175
5.1	Introduction	175
5.2	Running the memory dumper	175
5.2.1	Example of use	176
5.3	Error messages	176
6	iemit — memory interface configurer	177
6.1	Introduction	177
6.2	Running iemit	178
6.3	Output files	180
6.4	Interactive operation	180
6.4.1	Page 0	180
6.4.2	Page 1	182
6.4.3	Page 2	186
6.4.4	Page 3	187
6.4.5	Page 4	188
6.4.6	Page 5	189
6.4.7	Page 6	190
6.5	iemit error and warning messages	191
6.6	Memory configuration file	192
7	ieprom — ROM program convertor	195
7.1	Introduction	195
7.2	Prerequisites to using the ieprom tool	196
7.3	Running ieprom	196
7.3.1	Examples of use	197
7.4	ieprom control file	197
	Statement	198
	Parameter/Description	198
	Statement	200
	Parameter/Description	200
7.5	What goes into the EPROM	200
7.5.1	Memory configuration data	200
7.5.2	Parity registers	201
7.5.3	Jump instructions	201
7.5.4	Bootable file	202
7.5.5	Traceback information	202
7.6	ieprom output files	202
7.6.1	Binary output	202
7.6.2	Hex dump	202
7.6.3	Intel hex format	203
7.6.4	Intel extended hex format	203

7.6.5	Motorola S-record format	203
7.7	Block mode	203
7.7.1	Memory organization	203
7.7.2	When to use block mode	204
7.7.3	How to use block mode	204
7.8	Example control files	205
7.8.1	Simple output	205
7.8.2	Using block mode	205
7.9	Error and warning messages	206
8	ilibr — librarian	207
8.1	Introduction	207
8.2	Running the librarian	208
	Example	208
8.2.1	Default command line	208
8.2.2	Library indirect files	209
8.2.3	Linked object input files	209
8.2.4	Library files as input	209
8.3	Library modules	209
8.3.1	Selective loading	210
8.3.2	How the librarian sorts the library index	210
8.4	Library usage files	210
8.5	Building libraries	211
8.5.1	Rules for constructing libraries	211
8.5.2	General hints for building libraries	211
8.5.3	Optimizing libraries	211
	All libraries	212
	Libraries containing occam modules	212
	Semi-optimized library build targeted at all transputer types	212
	Optimized library targeted at all transputer types ..	213
	Library build targeted at specific transputer types ..	213
8.6	Error Messages	214
8.6.1	Warning messages	214
8.6.2	Serious errors	214
9	ilink — linker	217
9.1	Introduction	217
9.2	Running the linker	218
9.2.1	Default command line	219
9.3	Linker indirect files	219
9.3.1	Linker indirect files supplied with the toolset	220
9.4	Linker directives	220

9.4.1	#alias basename { aliases }	220
9.4.2	#define symbolname value	221
9.4.3	#include filename	221
9.4.4	#mainentry symbolname	221
9.4.5	#reference symbolname	221
9.4.6	#section name	222
9.5	Linker options	222
9.5.1	Processor types	222
9.5.2	Error modes – options H, S and X	223
9.5.3	TCOFF and LFF output files – options T, LB, LC	223
9.5.4	Extraction of library modules – option EX	224
	Example: Extraction from a user library	224
	Example: Extraction from a user library, using the run-time library	225
	Example: Extraction from a user library, for multiple processor types	225
	Example: Generation of a completely linkable library	226
	Extraction using #define	226
9.5.5	Display information – option I	227
9.5.6	Virtual memory – option KB	227
9.5.7	Main entry point – option ME	227
9.5.8	Link map filename – option MO	227
9.5.9	Linked unit output file – O	227
9.5.10	Permit unresolved references – option U	228
9.5.11	Disable interactive debugging – Y	228
9.6	Selective linking of library modules	228
9.7	The link map file	228
9.7.1	MODULE record:	229
9.7.2	SECT record:	229
9.7.3	MAP record:	229
9.7.4	Value record:	230
9.8	Using imakef for version control	230
9.9	Error messages	230
9.9.1	Warnings	230
9.9.2	Errors	231
9.9.3	Serious errors	232
9.9.4	Embedded messages	235
10	ilist - binary lister	237
10.1	Introduction	237
10.2	Data displays	237
10.2.1	Modular displays	238
10.2.2	Example displays used in this chapter	238
10.3	Running the binary lister	238

10.3.1	Options to use for specific file types	239
10.3.2	Output device	240
10.3.3	Default command line	240
10.4	Specifying an output file – option O	240
10.5	Symbol data – option A	241
10.5.1	Specific section attributes	241
10.5.2	General symbol attributes	241
10.5.3	Example symbol data display	242
10.6	Code listing – option C	242
10.6.1	Example code listing display	243
10.7	Exported names – option E	244
10.7.1	Example exported names display	244
10.8	Hexadecimal/ASCII dump – option H	244
10.8.1	Example hex dump display	245
10.9	Module data – option M	245
10.9.1	Example module data display	246
10.10	Library index data – option N	246
10.10.1	Example library index display	247
10.11	Procedural interface data – option P	247
10.11.1	Example procedural data display	247
10.12	Specify reference – option R	248
10.13	Full listing – option T	248
10.13.1	Example full data display	248
10.13.2	Configuration data files	249
10.14	File identification – option W	249
10.14.1	Example file identification display	250
10.15	External reference data – option X	251
10.15.1	Example external reference data display	251
10.16	Error messages	251
10.16.1	Warning messages	252
10.16.2	Serious errors	252
11	imakef — makefile generator	253
11.1	Introduction	253
11.2	How imakef works	253
11.3	File extensions for use with imakef	254
11.3.1	Target files	254
11.4	Linker indirect files	257
11.5	Library indirect and library usage files	257
11.6	Running the makefile generator	257
11.6.1	Example of use	258
11.6.2	Specifying language mode	259

11.6.3	Configuration description files	259
11.6.4	Disabling debug data	259
11.6.5	Removing intermediate files	260
11.6.6	Files found on ISEARCH	260
11.6.7	Map file output for imap	260
11.7	imakef examples	260
11.7.1	C examples	261
	Single transputer program	261
	Multitransputer program	262
11.7.2	occam examples	263
	Single transputer program	263
	Multitransputer program	264
11.7.3	Mixed language program	265
11.8	Format of makefiles	266
11.8.1	Macros	266
11.8.2	Rules	266
	Example:	266
	Action strings	267
11.8.3	Delete rule	267
11.8.4	Editing the makefile	267
	Adding options	267
	Re-running imakef	267
11.9	Error messages	268
12	imap — memory mapper	271
12.1	Introduction	271
12.2	Running the map tool	272
12.2.1	Source files required by imap	273
12.2.2	Re-directing imap's output	274
12.3	Output format	274
12.3.1	imap memory map structure	275
12.3.2	Process types	276
12.3.3	User processes	276
12.3.4	Module memory usage	276
12.3.5	Other processes	277
12.3.6	Symbol table	277
12.4	Example	278
12.5	Error messages	281
12.5.1	Serious errors	281
12.5.2	Fatal errors	281
13	iserver — host file server	283
13.1	Introduction	283
13.2	Loading programs	283

13.3	Host interface	283
13.4	Access to transputer networks	284
	User links	284
	The session manager	284
13.5	Running the iserver	284
13.5.1	Examples of use	285
13.5.2	Server environment variables	286
13.5.3	Loading programs	286
	Running a program using the iserver — option SB .	286
	Sending data down a user link — option SC	286
	Running programs which do not use the server ...	286
	Analyzing a transputer network — option SA	287
	Terminating the server	287
13.5.4	Supplying parameters to a program	287
13.5.5	Specifying the transputer resource — option SL ...	287
13.5.6	Terminating on error — option SE	287
13.5.7	Terminating the server	287
13.5.8	Specifying the session manager configuration file .	288
13.6	Using the session manager interface	288
13.6.1	Session manager commands	288
13.6.2	The options command	288
13.6.3	The iserver command	289
13.6.4	User-defined commands	290
	Running the debugger from the session manager .	291
13.6.5	Host OS commands	291
13.7	Connecting transputers to computer networks	291
13.7.1	Capabilities	291
13.7.2	The connection database	292
13.7.3	Using a specific node	293
13.8	The connection database	293
13.8.1	Connection databases	293
	Capability names	294
13.8.2	Connection database format	294
13.8.3	Example connection databases	295
	PC development system	295
	Sun workstation	296
	IMS B300	296
13.9	New server features	297
13.9.1	Session manager	297
13.9.2	Connection manager	297
13.9.3	New command line options	297
13.9.4	User interrupt	297
13.9.5	Exit codes	298
13.9.6	Error codes	298
13.9.7	Stream identifier validation	298

13.9.8	Record structured file support	298
13.10	Error messages	298
13.10.1	Additional error messages	300
14	isim — T425 simulator	303
14.1	Introduction	303
14.2	Running the simulator	303
14.2.1	Passing in parameters to the program	304
14.2.2	Example of use	305
14.2.3	ITERM file	305
14.3	Monitor page display	305
14.4	Simulator commands	306
14.4.1	Specifying numerical parameters	306
14.4.2	Keys mapped by ITERM	306
14.4.3	Command summary	307
14.4.4	Command descriptions	307
14.5	Batch mode operation	313
14.5.1	Setting up ISIMBATCH	313
14.5.2	Input command files	313
14.5.3	Output	313
14.5.4	Batch mode commands	313
14.6	Error messages	314
15	iskip - skip loader	317
15.1	Introduction	317
15.1.1	Uses of the skip tool	317
15.2	Running the skip loader	318
15.2.1	Skipping a single transputer	319
	Subsystem wired down:	319
	Subsystem wired subs:	319
15.2.2	Skipping multiple transputers	319
15.2.3	Loading a program	320
15.2.4	Monitoring the error status – option E	320
15.2.5	Clearing the error flag	321
15.3	Error messages	321
	Appendices	323
A	Toolset conventions and defaults	325
A.1	Command line syntax	325
A.1.1	General conventions	325
A.1.2	Standard options	325
A.2	Unsupported options	326

A.3	Filenames	326
A.4	Search paths	326
A.5	Standard file extensions	327
A.5.1	Main source and object files	328
A.5.2	Indirect input files (script files)	329
A.5.3	Files read by the memory map tool imap	329
A.5.4	Other output files	329
A.5.5	Miscellaneous files	330
A.6	Extensions required for imakef	330
A.7	Message handling	331
A.7.1	Message format	331
A.7.2	Severities	331
A.7.3	Runtime errors	332
B	Transputer types and classes	333
B.1	Transputer types supported by this toolset	333
B.2	Transputer types and classes	333
B.2.1	Single transputer type	333
B.2.2	Creating a program which can run on a range of transputers	334
B.2.3	Linked file containing code compiled for different targets	335
B.2.4	Classes/instruction sets – additional information ..	337
B.3	Transputer type command line options	339
C	Using the assembler	341
C.1	Introduction	341
C.2	Running the assembler	341
C.2.1	Specifying the source filename	341
C.2.2	Use of icc command options with the assembler ..	342
C.2.3	Using the pre-processor with the assembler	342
C.3	Language	343
C.3.1	Label definitions	343
C.3.2	Symbols	343
C.3.3	Expressions	343
C.3.4	Transputer instruction mnemonics	345
C.3.5	Comments	345
C.4	Assembler directives	346
C.5	BNF grammar for assembler language	376
C.6	Errors	379
C.6.1	Fatal Errors	379
C.6.2	Serious Errors	379

C.6.3	Errors	380
D	iserver protocol	383
D.1	iserver packets	383
D.2	Server commands	383
D.3	File commands	385
D.3.1	Fopen – Open a file	385
D.3.2	Fclose – Close a file	386
D.3.3	Fread – Read a block of data	387
D.3.4	Fwrite – Write a block of data	387
D.3.5	FGetBlock – Read a block of data and return success	388
D.3.6	FPutBlock – Write a block of data and return success	389
D.3.7	Fgets – Read a line	389
D.3.8	Fputs – Write a line	390
D.3.9	Fflush – Flush a stream	390
D.4	Record Structured file commands	391
D.4.1	FopenRec – Open a record structured file	391
D.4.2	FGetRec – Read a record	393
D.4.3	FPutRec – Write a record	393
D.4.4	FputEOF – Write an end of file record	394
D.4.5	Fseek – Set position in a file	394
D.4.6	Ftell – Find out position in a file	395
D.4.7	Feof – Test for end of file	395
D.4.8	Ferror – Get file error status	396
D.4.9	Remove – Delete a file	396
D.4.10	Rename – Rename a file	397
D.4.11	Isatty – Discover if a stream is connected to a terminal	397
D.4.12	FileExists – Check to see if a file exists	398
D.4.13	FerrStat – Get file error status	398
D.5	Host commands	399
D.5.1	Getkey – Get a keystroke	399
D.5.2	Pollkey – Test for a key	399
D.5.3	RequestKey – Request a single keyboard 'event'	400
D.5.4	Getenv – Get environment variable	400
D.5.5	Time – Get the time of day	401
D.5.6	System – Run a command	401
D.5.7	Translate – Translate an environment variable	402
D.6	Server commands	403
D.6.1	Exit – Terminate the server	403
D.6.2	CommandLine – Retrieve the server command line	403
D.6.3	Core – Read peeked memory	404

D.6.4	Version – Find out about the server	405
D.6.5	GetInfo – Obtain information about the host and server	406
D.6.6	CommandArgs – Retrieve the server command line arguments	407
D.7	Reserved Tags and Third Party Tags	408
D.7.1	MSDOS – Perform MS–DOS specific function	408
D.7.2	SocketA – make a socket library call	409
D.7.3	SocketM – make a socket library call	409
D.7.4	ALSYS – Perform Alsys specific function	409
D.7.5	KPAR – Perform Kpar specific function	410
D.8	Record Structured file format	410
D.8.1	SunOS and MS–DOS	410
	Formatted Sequential	410
	Unformatted Sequential	410
	Formatted Direct	410
	Unformatted Direct	410
D.9	Termination codes	411
E	ITERM files	413
E.1	Introduction	413
E.2	The structure of an ITERM file	413
E.3	The host definitions	414
E.3.1	ITERM version	414
E.3.2	Screen size	414
E.4	The screen definitions	414
E.4.1	Goto X Y processing	415
E.5	The keyboard definitions	416
E.6	Setting up the ITERM environment variable	416
E.7	Iterms supplied with a toolset	417
E.8	An example ITERM	418
F	Bootstrap loaders	421
F.1	Introduction	421
F.1.1	The example bootstrap	421
	Transfer of control	422
F.1.2	Writing bootstrap loaders	422

Preface

Host versions

The documentation set which accompanies the ANSI C toolset is designed to cover all host versions of the toolset:

- IMS D7314 – IBM PC compatible running MS-DOS
- IMS D4314 – Sun 4 systems running SunOS.
- IMS D6314 – VAX systems running VMS.

About this manual

This manual is the *Toolset Reference Manual* to the ANSI C toolset.

The manual provides reference material for each tool in the toolset describing:

- Command line syntax, including an example command line.
- Command line options.
- How to run the tool.
- A list of error messages which may be obtained.

Many of the tools in the toolset are generic to other INMOS toolset products i.e. the occam and FORTRAN toolsets and the documentation reflects this. Examples are given in C.

The appendices provide details of:

- Toolset conventions.
- Transputer types.
- The C assembler.
- Server protocol.
- ITERM files.
- Bootstrap loaders.

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 345 01 ANSI C Toolset User Guide*

Describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two sections; '*Basics*' which describes each of the main stages of the development process and includes a '*Getting started*' tutorial. The '*Advanced Techniques*' section is aimed at more experienced users. The appendices contain a glossary of terms and a bibliography. Several of the chapters are generic to other INMOS toolsets.

- *72 TDS 346 01 ANSI C Toolset Reference Manual (this manual)*
- *72 TDS 347 01 ANSI C Language and Libraries Reference Manual*

Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details are also provided about how to modify the run-time startup system, although only the very experienced user should attempt this.

- *72 TDS 348 01 ANSI C Optimizing Compiler User Guide*

Provides reference and user information specific to the ANSI C optimizing compiler. Examples of the type of optimizations available are provided in the appendices. This manual should be read in conjunction with the reference chapter for the standard ANSI C compiler, provided in the *Tools Reference Manual*.

- *72 TDS 354 00 Performance Improvement with the DX314 ANSI C Toolset*

This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual*. **Note:** details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide*.

- *72 TDS 355 00 ANSI C Toolset Handbook*

A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual*.

- *72 TDS 360 00 ANSI C Toolset Master Index*

A separately bound master index which covers the *User Guide*, *Toolset Reference Manual*, *Language and Libraries Reference Manual*, *Optimizing Compiler User Guide* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.

occam and FORTRAN toolsets

At the time of writing the occam and FORTRAN toolset products referred to in this document set are still under development and specific details relating to them are subject to change. Users should consult the documentation provided with the corresponding toolset product for specific information on that product.

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
Teletype	Used to distinguish command line examples, code fragments, and program listings from normal text.
<i>Italic type</i>	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote optional items in command syntax.
Brackets []	Used in command syntax to denote optional items on the command line.
Ellipsis . . .	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
	In command syntax, separates two mutually exclusive alternatives.

Tools

1 `icc` — ANSI C compiler

This chapter describes in detail the ANSI C compiler `icc`. It describes the command line syntax, compiler options, preprocessor directives, and other features of the compiler such as support for transputer code. The chapter ends with a list of error messages.

This chapter applies to both the standard and optimizing C compilers supplied with this toolset unless otherwise stated. The '*ANSI C Optimizing Compiler User Guide*' provides details of command line options only available when using the optimizing compiler and also provides a complete list of error diagnostics for the optimizing compiler.

1.1 Introduction

The ANSI C compiler conforms fully with the X3.159–1989 ANSI standard for the C programming language. This standard has now been ratified as "ISO/IEC 9899:1990 Programming languages — C". The ANSI C compiler provides support for concurrent programming as well as some additional extensions to the C language including compiler directives, pragmas and low level programming.

The ANSI standard for the C language defines the language including runtime library support, new types and function prototyping. For a summary of the differences between ANSI C and the original definition of the language see chapter 4 '*New features in ANSI C*' in the accompanying *Language and Libraries Reference Manual*. The ANSI C compiler includes support for parallel programming through a set of library functions with associated types and structures, a mechanism for incorporating transputer code sequences, and a group of compiler pragmas for enabling compiler options in sections of code and for conveying directives to the linker. The transputer code mechanism supports the full set of transputer instructions and operations and also supports labels.

Parallel processing is achieved through a library of process, channel, and semaphore functions and their related types and data structures. Calls to the functions are compiled by `icc` into highly efficient parallel code for the transputer.

`icc` generates code for a particular transputer, transputer type, or class, and a target should be specified for all compilations. The default is to produce code for the IMS T414.

1.2 Running the compiler

To invoke the compiler use the following command line:

► **icc** *filename* {*options*}

where: *filename* is the C program source code. If no extension is given .C is assumed. Only one filename may be given on the command line.

options is a list of options given in table 1.1. Options to select the transputer target for the compilation are listed in appendix B.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order on the command line.

Options must be separated by spaces.

If no arguments are given on the command line brief help information is displayed; the full help page is displayed by using command option '**HELP**'.

Note: **icc** must be invoked in a writeable directory, that is, one in which you (or any alias you use to invoke the compiler) have *write* access.

Option	Description
<i>Transputer type</i>	See appendix B for a list of options to specify transputer type.
AS	Assemble the input file to produce an object file. The compiler phase is suppressed. See section 1.2.7.
C	Performs a syntax check only. Generates no object code. This option is ignored by the optimizing compiler.
D symbol	Defines a symbol. Same as #define symbol 1 at the start of the source file.
D symbol=value	Defines a symbol and assigns a value. Same as #define symbol value at the start of the source file.
EC	Disables checks for invalid type casts. ANSI compliance check.
EP	Disables checks for invalid text after #else or #endif . ANSI compliance check.
EZ	Disables checks for zero-sized arrays. ANSI compliance check.
FC	Change the signedness property of plain char to be signed. The default is to compile chars as unsigned.
FH	Performs a number of software quality checks. See section 1.2.9.
FM	Generates warning messages on #defined but unused macros.

Table 1.1 Standard **icc** compiler options

Option	Description
FS	Directs the compiler to treat right-shifts of signed integers as arithmetic shifts. See section 1.2.8.
FV	Reports all externally visible functions and variables which are declared but unreferenced, and have file scope.
G	Generates comprehensive debugging data. The default is to produce minimal debugging data. Debugging data is required for the correct operation of <code>idebug</code> .
HELP	Displays full help information for the tool.
I	Displays detailed progress information at the terminal as the compiler runs.
J dir	Adds <i>dir</i> to the list of directories to be searched for source files incorporated with the <code>#include</code> directive in extended search paths. See section 1.4.9 for details.
KS	Enables stack checking.
O outfile	Specifies an output file. If no filename is given the compiler derives the output filename from the input filename stem and adds the <code>.tco</code> extension.
p mapfile	Produces a map of workspace for each function defined in the file, and a map of the static area of the whole file. The map is written to the file <i>mapfile</i> . See section 1.3.
PP	Lists the preprocessed source file to <code>stdout</code> .
S	Compiles the source file to assembly language and writes it to a file. Assembly is suppressed and no object code is produced. The file is named after the input file and given the <code>.s</code> extension.
U symbol	Disables a symbol definition. Equivalent to <code>#undef symbol</code> at the start of the source file.
WA	Suppresses messages warning of '=' in conditional expressions.
WD	Suppresses messages warning of deprecated function declarations.
WF	Suppresses messages warning of implicit declarations of <code>extern int()</code> .
WN	Suppresses messages warning of implicit narrowing or lower precision.
WT	Suppresses messages warning of the possibility of less efficient code when compiled for a transputer class.
WV	Suppresses messages warning of non-declaration of <code>void</code> functions

Table 1.1 continued – Standard `icc` compiler options

Examples of use:**UNIX based toolsets:**

```
icc hello
ilink hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS/VMS based toolsets:

```
icc hello
ilink hello.tco /f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

1.2.1 Optimizing compiler options

There are a number of options which may be specified on the command line but which the standard ANSI C compiler will ignore. These options are supported exclusively by the optimizing ANSI C compiler and they enable and support a range of code optimizations performed at compile time. These options are documented in detail in the 'ANSI C Optimizing Compiler User Guide' which accompanies this toolset. The options are listed in table 1.2 for completeness.

Option	Description
FSC	Provides information on how the compiler has treated routines with respect to side effects.
O0	Disable optimization.
O1	Enable local optimization.
O2	Enable both global and local optimization.
QS	Optimize for space.
QT	Optimize for time.
WS	Suppress warning messages about possible side effects.

Table 1.2 Optimizing compiler options

1.2.2 Transputer targets

The compiler generates code for a specific transputer type. This means that a processor type should be specified for all transputer targets except the default which is built into the compiler. The default processor type which is used if no target is specified is the T414.

Transputers are also grouped into classes for the purpose of generating common code suitable for running on a number of different transputer targets. Transputer classes group transputers according to word size and instruction set compatibility. They can be used to generate code for combinations of transputers.

The use of transputer types and classes in developing programs is explained in appendix B. The command line options for selecting a transputer target are given in this appendix.

1.2.3 Error modes

All code in mixed language transputer programs must be compiled and linked in the same or a compatible error mode. `icc` always generates code in UNIVERSAL error mode, which is compatible with HALT and STOP error modes created by other INMOS compiler toolsets.

The error mode for a mixed language program can be consolidated into a single mode for the entire program by specifying the appropriate linker option. If no mode is specified the linker generates the program in HALT mode.

1.2.4 Default command line options

Commonly used command line parameters can be defined in the host environment variable `ICCARG`. Parameters specified in this way are automatically added to the end of the command line when the compiler is invoked.

Command line parameters must be specified in `ICCARG` using the syntax required by the `icc` command line.

1.2.5 File extension defaults

The `.c` extension is assumed on input source files and does not need to be specified. If no output file is specified the compiled object file is named after the input file and given a `.tco` extension. A `.tco` extension is also added if a file is specified without an extension.

When the input file is an assembler source file, a file extension (other than `.c`) *must* be specified. Even though the `'AS'` option is specified to invoke the assembler, the compiler will assume a C source file is to be compiled if a file extension is not specified on the command line.

1.2.6 Search paths

The normal search paths are used for locating files specified on the command line. The search rules are described in appendix A.

Search paths for files imported with the `#include` compiler directive differ slightly from those for files specified on the command line and can be extended by the use of special syntax and a command line option. Details of this facility can be found in section 1.4.9.

1.2.7 Using the assembler

Assembler source files may be assembled by using the `icc` command line option `'AS'`. This causes the compilation phase of the compiler to be suppressed and the input file to be passed directly to the assembler. If the input assembly source file

contains preprocessor directives, the compiler preprocessor must first be used to process the source file; the output from the preprocessor may then be used as input to the assembler.

The use of the assembler is described in appendix C, together with examples of how it is invoked. The file name conventions for assembler files and the command options which may be used with the assembler are listed. The appendix also describes the syntax of assembler directives and lists the error messages which may be generated by the assembler.

1.2.8 Compatibility with other C implementations

Two compiler options are provided which may assist users porting existing C code to transputer systems.

Arithmetic right shifts

By default, the compiler implements right-shifts of signed integers as logical shifts, the command line option **FS** switches the implementation. This allows correct working of programs which assume that right shifts of signed values propagate the sign.

Signedness of `char`

By default the compiler implements plain `chars` as **unsigned chars**. The command line option **FC** switches the implementation to **signed char**. Details of type representation are given in chapter 6 of the *ANSI C Language and Libraries Manual*.

1.2.9 Software quality check

The **FH** option allows policing of software quality requirements. The option requires all externally visible definitions to be preceded by a declaration (from a header file), thus guaranteeing consistency.

When the **FH** option is used the compiler reports:

- all forward **static** declarations which are unused when the function is defined.
- all repeated macro definitions (this is when macros are redefined to the same value; redefining a macro to a different value is always diagnosed as an error).
- (optimizing compiler only) — reports all unused function arguments.

Note: the standard compiler reports all unused function arguments by default.

1.3 Memory map

The compiler may be instructed, via the `-Z mapfile` option, to produce a map of workspace for each function defined in the file, and a map of the static area of the whole file. The file contains information which may assist the user during program debugging. The map is written to the file *mapfile*.

The file consists of a series of workspace maps; one for each routine, giving details of workspace requirements. These are followed by a series of section maps; one for each section of code, listing details of its static variables.

The file is generated in text format and is structured as follows:

- The name of the source file for which the map of code and data is being produced. The full pathname will be given if it exists.
- Version data for the compiler.
- The target transputer of the compilation, T805, T400 etc.
- The error mode of the compilation, this is always UNIVERSAL for C programs.
- Name of the routine for which the map of workspace is being produced. Items in the workspace map are given in ascending order of workspace offset.
 - List of local variables giving their offset (in bytes) into the routine's workspace. This list may include temporary variables introduced by the compiler.
 - List of formal parameters giving their name and offset (in bytes) into the routine's workspace. Parameters added by the compiler may also be listed, see table 1.3.
 - The workspace requirement of the routine in bytes. **Note:** this includes the four word call overhead introduced by the transputer call instruction.
- Name of the section for which the section map is being produced. Items in the section map are given in ascending order of section offset.
 - A list of static variables or routines, giving the following details:
 - Name of static variable or routine. This may be in the form '*<name>%xp*', see table 1.4
 - Type of variable or routine
 - Offset in bytes into static data or code area
 - Other properties of variable or routine, see table 1.4.

Static variables are either placed in the static or code areas. Details of how the compiler allocates space for static data are given in section 6.15 of the *ANSI C Language and Libraries Manual*.

Formal parameter
Compiler temporary
Result pointer
Return address
Global static base pointer (gsb)
Static link

Table 1.3 Parameters inserted by compiler

Property	Description
global	Globally visible static item.
static	Static item.
pointer to external object	Static item introduced by the compiler to enable code to access an external object. The name of the external object is used as the prefix to the compiler generated name. e.g. 'fred* xp ' is a static item introduced by the compiler which points to an external object named 'fred'.
translated from data name	Static items whose name has been modified by the IMS_translate pragma are listed under the name that is put into the object file. They are annotated with the message: 'translated from <i>sourcename</i> ', where <i>sourcename</i> is the name used in the source file.

Table 1.4 Static variable properties

Note: The message "No local variables" may be displayed if no user variables are found, however, compiler temporaries may have been assigned to workspace. In addition some compiler temporaries may not be listed in the map file.

The compiler does not generate an explicit "No static data" message. If a file does not contain static data, such information will not be present in the map file.

Information generated in the compiler map file may be extracted by the **imap** tool. This tool can be used to produce a memory map for the program after it has been compiled, linked and collected. See chapter 12.

```

Map of code and data for source file hello.c
=====
Created by INMOS C compiler Version 2.02.05 (built at 18:11:13 Dec 10 1991)

Target processor : T4
Error mode       : UNIVERSAL

Map of workspace
-----
Routine : main

Variable name          Offset (bytes)
b                      0
a                      4

Formal parameter name  Offset (bytes)
<return address>      8
<gsb>                 12

Workspace size = 24 bytes

Map of workspace
-----
Routine : bill

Variable name          Offset (bytes)
<compiler temporary>  0
<compiler temporary>  4
z                     12

Formal parameter name  Offset (bytes)
<return address>      20
<gsb>                 24
<result pointer>      28
c                     32
f                     36

Workspace size = 36 bytes

Section map
-----
Section name : static%base

Name          Type          Offset (bytes)
fred          static data  0

Section map
-----
Section name : text%base

Name          Type          Offset (bytes)
main          code          4          global
bill         code          36         static
-----

```

Figure 1.1 Example compiler map

1.4 Compiler directives

1.4.1 #define

Syntax: **#define** *name* [(*arg1*, ..., *argn*)] [*value*]

#define allows simple macro substitution to be performed. In its simplest mode of operation *name* and *value* represent a series of ASCII characters causing the preprocessor to substitute all occurrences of *name* by *value* (which may be null). Arguments may also appear after the name, and when this happens the preprocessor will still replace all occurrences of *name* and its following arguments by *value*, but in this case the value string will have been defined in terms of the expected arguments, and will therefore exhibit a dependence on the original text.

```
#define YES 1      /* replace all occurrences
                  of YES by 1 */

#define max(a,b) (a > b ? a : b)
/* max(2,4) will be replaced by
   (2 > 4 ? 2 : 4) */
```

1.4.2 #elif *constant_expression*

Syntax: **#elif**

This directive can be used in place of the sequence

```
#else
#if constant_expression.
```

1.4.3 #else

Syntax: **#else**

This directive can be used with the **#if**, **#ifdef**, and **#ifndef** directives to mark the beginning of text which will be ignored whenever the expression following the **#if** evaluates to a non-zero value.

1.4.4 #endif

Syntax: **#endif**

This directive must be used with the **#if**, **#ifdef**, and **#ifndef** directives to mark the end of the text which may be affected by the **#if ... #else ... #endif** construct.

1.4.5 `#error`

Syntax: **`#error`** *text*

This directive causes an explicit error with the text following the directive displayed in the error message. This is useful for determining which pieces of code are being bypassed by a construct of the form **`#if ... #else ... #endif`**.

1.4.6 `#if`

Syntax: **`#if`** *constant_expression*

This directive, along with the **`#else`** and **`#endif`** directives, is used in a similar way to the `if ... else` construct of many high level programming languages. When it is encountered, the preprocessor evaluates the following constant expression and if it is zero it ignores all text up to the following **`#else`** or **`#endif`** directive. If, however, the expression evaluates to non-zero, then the text between the **`#else`** and **`#endif`** directives (if any) is ignored. This mechanism would typically be used to allow conditional compilation.

As an extension to this directive, the preprocessor also allows 'if defined' type expressions. In this case 'defined' is used as a unary operator which returns true if its operand represents an identifier that is currently defined within the preprocessor's symbol table, and false if it is not. By combining this operator with the logical operators it is possible to build complex expressions of the form:

```
#if defined foo & ! defined dummy  
  
      /* if foo is defined & dummy is not */
```

1.4.7 `#ifdef`

Syntax: **`#ifdef`** *identifier*

This directive works in a similar way to the **`#if`** directive, but instead of basing its decision on the result of an expression it uses the existence or non-existence of the identifier within the preprocessor's symbol table as the criterion. If the identifier has not previously appeared in a **`#define`** directive or if it is not one of the predefined identifiers then all text up to the following **`#else`** or **`#endif`** directive is ignored; otherwise all text between the **`#else`** and **`#endif`** directives is ignored.

1.4.8 `#ifndef`

Syntax: **`#ifndef`** *identifier*

This directive is similar to **`#ifdef`**, except that the text is passed if *identifier* is not currently defined.

1.4.9 #include

Syntax: **#include** *filename*

The **#include** directive instructs the preprocessor to read the contents of the named file as if they were at the current position in the current file. The filename must be enclosed within angle brackets (*<filename>*) or double quotes ("*filename*"). The two forms generate different search strategies.

If angle brackets are used *only* those directories specified by **ISEARCH** are searched. No other directories (including the current directory) are searched. This method is mainly used to include the standard library header files.

If double quotes are used to enclose the filename the standard toolset search is used, but incorporating a method for extending the search list. First the current directory is searched. If the file is not found the search continues with the list of directories specified after the compiler '**J**' option. If the file is still not found, or if no list is given, directories specified by **ISEARCH** are searched.

A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file. There is no fixed limit to **#include** nesting.

Relative directory names

Relative directory names are treated as relative to the directory containing the current source file.

Backslash character in filenames

In included filenames the backslash is not treated as introducing an escape sequence unless it is followed by another backslash ('\\').

1.4.10 #line

Syntax: **#line** *linenumber* [*filename*]

This directive instructs the compiler that subsequent lines begin with line number *linenumber* in the file *filename*. If no file name is specified, the original name is retained. *linenumber* must be within the range 1 to 32767 inclusive.

1.4.11 #pragma

Syntax: **#pragma** *pragma* (*params*)

This directive activates and deactivates various compiler options in sections of C code. It may be used to set (or override) options specified on the command line. Most pragmas also take parameters or numerical arguments.

Table 1.5 lists the main compiler pragmas and table 1.6 lists the parameters to **IMS_on** and **IMS_off**.

Option	Description
IMS_on (<i>params</i>)	Enables specific compiler actions. Takes a list of parameters which specify the actions to be enabled.
IMS_off (<i>params</i>)	Disables specific compiler actions. Takes a list of parameters which specify the actions to be disabled.
IMS_nolink (<i>functionname</i>)	Compiles the function <i>functionname</i> without a global static base parameter. The function must already have been declared but must not have been defined or called. This pragma is used for importing code written using languages such as OCCam which do not use static data, and for exporting C functions to the same languages.
IMS_linkage ([<i>"name"</i>])	Enables the user to change the order in which code modules are linked together; this may aid the use of faster on-chip RAM. The compiler creates the object code into a section named <i>"text%base"</i> . The IMS_linkage pragma causes the compiler to change the name of the section to that supplied in the string. If no string is present, <i>"pri%text%base"</i> is used; this section being inserted at the front by the linker in the default case. A linkage command (see 9.4.6) controls of the ordering of the sections. The linkage directive should appear at the start of the code, before any function definitions.
IMS_modpatchsize (<i>n</i>)	Specifies the number of bytes reserved by the compiler for a linker module number patch. <i>n</i> has default values of 3 for 32-bit targets and 2 for 16-bit targets.
IMS_codepatchsize (<i>n</i>)	Specifies the number of bytes <i>n</i> reserved by the compiler for a linker code patch. <i>n</i> has a default value of 6 for 32-bit targets and 4 for 16-bit targets.
IMS_translate (<i>name</i> , <i>"newname"</i>)	The compiler replaces all references to <i>name</i> (e.g. an external routine) by <i>"newname"</i> . <i>"newname"</i> is a C string which can contain alphanumeric characters; the underscore ('_'), percent ('%'), or full stop ('.') characters.
IMS_descriptor (<i>function-name</i> , <i>language_type</i> , <i>work-space</i> , <i>vectorspace</i> , <i>"descriptor-string"</i>)	Creates a TCOFF descriptor for C functions. Further details are given below.

Table 1.5 **icc** compiler pragmas

Parameter	Short form	Description
<code>channel_pointers</code>	<code>cp</code>	Treats a variable of type <code>Channel</code> in the scope of the definition <code>typedef const volatile void *</code> as a channel type for the debugger. Default is off. This pragma is enabled in the header file <code>channel.h</code> . If <code>channel.h</code> is included in the program this pragma will remain active until specifically disabled.
<code>inline_ops</code>	<code>il</code>	Compiles certain operations on long operands (signed or unsigned) on 16-bit targets as in-line operations rather than as calls to the compiler library. Operations affected are: <code>~</code> (bitwise complement), <code>+</code> , <code>-</code> , <code>&</code> (bitwise AND), <code> </code> (bitwise OR), <code>^</code> (bitwise exclusive OR), <code><<</code> , <code>>></code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code> , <code>>=</code> , and <code>></code> . Default is on.
<code>printf_checking</code>	<code>pc</code>	Checks that arguments passed to a function conform to the format used by <code>printf</code> . Default is off. This pragma is normally used to check formal arguments which are to be passed directly as format strings to <code>printf</code> . For each function within the scope of the pragma the last formal parameter is read as a format string and subsequent variable arguments are checked for correct type, according to the formatting rules of <code>printf</code> . This pragma is enabled in <code>stdio.h</code> for the declaration of <code>printf</code> and related functions, and subsequently disabled.
<code>scanf_checking</code>	<code>sf</code>	Checks that arguments passed to a function conform to the format accepted by <code>scanf</code> . Default is off. Otherwise this pragma has the same effect <code>printf_checking</code> . This pragma is enabled in <code>stdio.h</code> for the declaration of <code>scanf</code> and related functions, and subsequently disabled.
<code>stack_checking</code>	<code>sc</code>	Checks for stack overflow at the start of each function. Default is off.
<code>warn_bad_target</code>	<code>wt</code>	Warns of inferior code generated for a transputer class rather than for a specific transputer target. Default is on.
<code>warn_deprecated</code>	<code>wd</code>	Warns of parameterless function declarations. Default is on.
<code>warn_implicit</code>	<code>wi</code>	Warns of undeclared functions. Default is on.

Table 1.6 Parameters to `IMS_on` and `IMS_off`

Pragma IMS_nolink

The pragma `IMS_nolink` enables C routines to call or be called from OCCAM and other languages.

Syntax: **#pragma IMS_nolink (fname)**

The following code uses the pragma to allow an OCCAM routine `OCCAMREALOP` to be called in a C program:

```
extern float OCCAMREALOP(const float x,
                        const int op,
                        const float y);
#pragma IMS_nolink (OCCAMREALOP)
:
:
float x, y, z;
z = OCCAMREALOP(x, op_add, y);
```

The following code allows the C function `max` to be called from occam:

```
extern int max(const int x, const int y);
#pragma IMS_nolink (max)
extern int max(const int x, const int y)
{ return x > y ? x : y; }
```

Note: functions which have had the `IMS_nolink` pragma applied may not be called through a pointer. The library routine `call_without_gsb` is supplied to allow a call through a pointer to a nolinked function.

Pragma IMS_descriptor

The pragma `IMS_descriptor` creates a TCOFF descriptor for C functions. It also causes the definition of two TCOFF symbols giving the workspace and vector-space requirements of the function. This pragma is of particular use when modifying the C runtime startup code, further details of which are given in chapter 3 of the *ANSI C Language and Libraries Reference Manual*. It is also applicable when making use of the dynamic loading facility provided in the C library (see chapter 2 of the *ANSI C Language and Libraries Reference Manual* and chapter 12 of the *ANSI C Toolset User Guide*).

Syntax: **#pragma IMS_descriptor (functionname, language_type, **
 workspace, vectorspace, "descriptor-string")

The parameters to the pragma are given in table 1.7.

<i>functionname</i>	Name of the C function to which the descriptor applies.
<i>language_type</i>	The language in which the descriptor string is written. The language is given as a keyword: unknown occam ansi_c fortran iso_pascal modula2 ada assembler occam_harness Alternatively the <i>descriptor-string</i> may be an empty string, however, a language type must still be given.
<i>workspace</i>	The amount of workspace required by the function. (Expressed as a number of words).
<i>vectorspace</i>	The amount of vector space required by the function. (Expressed as a number of words). This is usually '0' for C functions.
<i>"descriptor-string"</i>	This is the descriptor string itself. If the string is not empty then it must contain an OCCAM style function declaration equivalent to the C function prototype.

Table 1.7 Parameters to `IMS_descriptor`

The rules governing the use of this pragma are as follows:

- The function must be externally visible.
- The function must have been declared before the pragma appears.
- The function must not have been defined before the pragma appears.
- The pragma must appear in the same file in which the function is defined.
- Only one descriptor pragma can exist per function.
- No argument to the descriptor pragma can be the result of earlier preprocessor substitutions.

An example of the use of this pragma follows:

```
void centry(int bill);

#pragma IMS_descriptor(centry, occam, 32, 0, \
    "PROC centry(VAL INT bill)\n SEQ\n:")

void centry(int bill)
{
    /* function body */
}
```

This defines an OCCAM descriptor for the function `centry`. A requirement for 32 words of workspace and no vectorspace is also recorded in the descriptor. The syntax for the descriptor string is the standard syntax for OCCAM descriptors.

Note: type compatibility between the parameters in OCCAM and C is retained by following the rules given in the *ANSI C Toolset User Guide*, Mixed language programming chapter.

Example TCOFF output from the above can be obtained using the 't' option on the linker tool `ilist`, as follows:

```
00000080 SYMBOL EXP "centry" id: 4
...
00000092 SYMBOL EXP UNI "centry'ws" id: 5
0000009F SYMBOL EXP UNI "centry'vs" id: 6
000000AC DEFINE SYMBOL id: 5 32
000000B1 DEFINE SYMBOL id: 6 0
000000B6 DESCRIPTOR id: 4 lang: OCCAM
ws: 32 vs: 0
PROC centry (VAL INT bill)
  SEQ
:
```

1.4.12 #undef

Syntax: `#undef identifier`

This directive causes the current definition of *identifier* (as defined using the `#define` directive) to be deleted.

1.5 Compiler predefinitions

Certain macros which identify global information, and some function names, are automatically recognized by the compiler. Generally, these items can be referenced directly in C programs and do not need to be declared.

Note: Predefined variables `_lsb` and `_params` (see section 1.5.2) should be declared to avoid spurious warning messages being generated by the compiler.

1.5.1 Macro names

All predefined macro names defined by the ANSI standard are present; they are:

<code>__DATE__</code>	—	The current date.
<code>__FILE__</code>	—	Name of the current source file.
<code>__LINE__</code>	—	Line number of the current line of source.
<code>__STDC__</code>	—	A non-zero value if the implementation conforms to ANSI C.
<code>__TIME__</code>	—	The current time.

Details of the ANSI macros and the values they can take can be found in chapter 4 of the *ANSI C Language and Libraries Reference Manual*.

The following INMOS macro names are also defined:

<code>__CC_NORCROFT</code>	— Derived from the Norcroft C compiler.
<code>__ICC</code>	— INMOS C compiler.
<code>__PTYPE</code>	— Processor type.
<code>__ERRORMODE</code>	— Execution error mode.
<code>__SIGNED_CHAR__</code>	— Signedness of the plain <code>char</code> type, defined if the <code>icc 'FC'</code> command line option is used.

Details of the macros and the values they can take can be found in chapter 5 of the *ANSI C Language and Libraries Reference Manual*.

1.5.2 Other predefines

Two further names `_lsb` and `_params` are predefined by the compiler. They can be used in expressions in the same way as C variables. Both represent addresses which may be manipulated in low level programming and *must* be declared as follows:

```
extern volatile const void *_lsb;
```

```
extern volatile const void *_params;
```

`_lsb` is a pointer to the base of the compiled file's data area.

`_params` is a pointer to the base of the the current function's parameter block. It can be used to obtain low level information about a function's runtime code.

The following example illustrates how `_params` can be used to determine a function's return address, global static pointer, and workspace pointer.

```
void p()
{
    extern volatile const void *_params;
    typedef struct paramblock
    {
        void *return_address;
        void *gsb;
        int regparam1, regparam2;
    }
    paramblock;

    paramblock *pp = (paramblock *)_params;

    /* Return address is: pp->return_address
       global static base sb is: pp->gsb
       caller Wptr is: (void *) (pp + 1) */
}
```


1.6 Transputer inline code

INMOS C provides different levels of support for inlining transputer instructions:

- A special keyword `__asm` can be used to enclose sequences of transputer instructions into C programs. The `__asm` statement and how to use it is described in chapter 5 of the *ANSI C Language and Libraries Reference Manual*.
- A number of functions are supplied which can be compiled inline as transputer instructions, provided the appropriate header files are included in the source code. The inputs and outputs of the instructions are treated as parameters to and results from the functions.

1.6.1 Inlined functions

Each of the supplied functions is designed to allow access to a transputer instruction which is not directly accessible from the C source level. **Note:** however, that the automatic inlining will only occur if the appropriate header file has been incorporated in the source code by using the `#include` directive. The header files contain prototypes for the routines. Table 1.8 lists the functions, the instructions they support and the header file which is required.

Function	Instruction supported	Header file
BitCnt	<i>bitcnt</i>	<i>misc.h</i>
BitCntSum	<i>bitcnt</i>	<i>misc.h</i>
BitRevNBits	<i>bitrevnbits</i>	<i>misc.h</i>
BitRevWord	<i>bitrevword</i>	<i>misc.h</i>
BlockMove	<i>move</i>	<i>misc.h</i>
CrcByte	<i>crcbyte</i>	<i>misc.h</i>
CrcWord	<i>crcword</i>	<i>misc.h</i>
DirectChanIn	<i>in</i>	<i>channel.h</i>
DirectChanInChar	<i>in</i>	<i>channel.h</i>
DirectChanInInt	<i>in</i>	<i>channel.h</i>
DirectChanOut	<i>out</i>	<i>channel.h</i>
DirectChanOutChar	<i>outbyte</i>	<i>channel.h</i>
DirectChanOutInt	<i>outword</i>	<i>channel.h</i>
memcpy	<i>move</i>	<i>string.h</i>
Move2D	<i>move2dall</i>	<i>misc.h</i>
Move2DNonZero	<i>move2dnonzero</i>	<i>misc.h</i>
Move2DZero	<i>move2dzero</i>	<i>misc.h</i>
ProcGetPriority	<i>ldpri</i>	<i>process.h</i>
ProcReschedule	–	<i>process.h</i>
ProcTime	<i>ldtimer</i>	<i>process.h</i>
strcpy	–	<i>string.h</i>

Table 1.8 Inlined functions

Note: the 'DirectChan...' functions must not be used with virtual channels: section 6.3.1 of the *ANSI C Toolset User Guide*, discusses this.

Descriptions of all the functions are provided in the *ANSI C Language and Libraries Reference Manual*.

1.7 Compiler diagnostics

This section lists diagnostic error messages generated by `icc`. The section is introduced by descriptions of some standard terms which may be encountered in the message texts.

1.7.1 Message format

Diagnostic messages are displayed in the standard toolset format for error messages. Details of the standard can be found in appendix A.

1.7.2 Severities

Diagnostics are tagged with a severity level which indicates their effect on the compilation. Severity levels are the same as those used in the toolset standard but have slightly different meanings, which are described below.

Information messages provide the user with information about the functioning or performance of the tool. They do not indicate an error and no user action is required in response.

Warning severity diagnostics are generated whenever legal, but unorthodox programming styles are detected. Compilation is unaffected and object code is generated normally.

Error severity diagnostics are generated whenever the compiler detects a programming error from which it can recover. Compilation continues, but may abort if more errors are detected subsequently. No object code is generated.

Serious severity diagnostics are generated when programming errors are detected from which the compiler cannot recover. Compilation continues but code has been lost. No object code is generated.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your local INMOS distributor or field applications engineer.

Error, *Serious*, and *Fatal* diagnostic messages return error codes for handling by system MAKE programs and batch files.

1.7.3 Standard terms

This section explains some of the standard terms and notation used in compiler error messages.

abstract declarator

When using explicit casts or when passing an argument to `sizeof()`, a data type must be specified. This can be done by declaring an object of the correct type without specifying the name of the object. Declarations of this type are called abstract declarations, because they apply to no known object.

Examples of abstract declarations are:

```
(int) a = b;      /* 'int' is the abstract
                  declarator */

sizeof(int [3]); /* 'int [3]' is the abstract
                  declarator */
```

char

Stands for a single ASCII character.

context

Stands for a type, for example, 'character constant', 'integer constant', and 'string constant'.

deprecated declaration

This means that a function declaration is incomplete. Declarations should specify the type of the function and the type of each formal parameter. If there are no parameters then the function type `void` should be specified.

expression

Stands for a C expression.

filename

A file name.

function prototype

A function declaration which usually precedes the function definition. It declares the function's type and the types of its parameters.

identifier

A C identifier, for example, a variable or function name.

initializer

An initial value which is assigned to an object at the time of its declaration.

message string

The string which follows a compiler directive.

op

An operator. Valid operators include: "++", "--", ">", "<=", and the unary operators &, *, + and -.

store class

A C storage class. Valid classes are **static** or **extern**.

string

Any string of ASCII characters.

struct/union

A variable of type **struct** or **union**.

type

A type identifier.

void context

This can occur at any point in a program where a value is not expected, for example, calling a function without using the returned number.

instruction

A transputer instruction, or a pseudo-instruction as accepted by the `__asm` construct.

1.7.4 ANSI trigraphs

The ANSI specification includes a number of three character sequences that can be used to represent certain ASCII characters that may not be present on all keyboards. These sequences, known as **trigraphs**, are used in compiler error messages to stand for these characters.

ANSI standard trigraph sequences consist of a sequence of 2 question marks followed by a third character. A complete list of ANSI trigraphs is given in the chapter 4 of the accompanying *ANSI C Language and Libraries Manual*.

1.7.5 Warning diagnostics

#define macro *identifier* defined but not used

The named macro has been defined, but not referenced in the rest of the program. This message is only generated if specifically enabled by the '**FM**' compiler option.

'&' unnecessary for function or array *identifier*

A pointer to a function or array is implied by use of the name alone; the '&' operator is not required.

'int identifier ()' assumed – 'void' intended?

A function was defined without specifying its type. The compiler assumes a function of type `int` if no type is specified.

***identifier* already has a descriptor defined, `pragma` ignored**

The `pragma IMS_descriptor` has already been applied to *identifier*; more than one application is invalid.

***identifier* has been called; `pragma` ignored**

The `pragma` must be applied to *identifier* before the latter has been called.

***identifier* has been defined; `pragma` ignored**

The `pragma` must be applied to *identifier* before the latter is defined.

***identifier* has not been declared; `pragma` ignored**

The `pragma` must be applied to *identifier* after the latter has been declared.

***identifier* is not a function; `pragma` ignored**

The argument to the `pragma` must be a function name.

***identifier* is not externally visible; `pragma` ignored**

The first argument to the `IMS_descriptor` `pragma` must be the name of an externally visible function.

***identifier* multiply translated, this translation ignored**

The `IMS_translate` `pragma` has been applied to *identifier* more than once.

***number* treated as *number* UL in 32-bit implementation**

No type was specified for the number. The compiler assumes `unsigned long` if no type was specified.

***op* : cast between function and object pointer**

The specified operator has been used in an expression involving pointers of different types, that is, a function pointer and an object pointer (a pointer to an area in memory).

***type identifier* declared but not used**

The named identifier has been declared, but not used in the program.

actual type *type* mismatches format '`%char`'

The type of an argument to `printf` or `scanf` does not match that implied by the control string.

ANSI '`char char char`' trigraph for '`char`' found – was this intended?

The specified three character sequence was found in the source program. This has been treated as an ANSI trigraph and substituted for the character shown.

argument and old-style parameter mismatch: *expression*

There is an old (non-prototype) style function definition in scope, and the type of an argument (after default argument promotion has taken place) does not agree with the type of the corresponding formal parameter.

Cannot delete temporary file *filename*

Host file system error.

Cannot generate stack check for *function* (*pragma nolinek* applied)

A stack check requires a static link, and the function *function* has been specified not to receive a static link (using *IMS_nolinek*). *icc* compiles the function with the stack check omitted.

character sequence */ inside comment**

The start-of-comment character sequence was detected within a comment. Check that the previous comment was terminated correctly.

Dangling 'else' indicates possible error

Within nested *if ... else* constructs, there is some ambiguity as to which 'if' relates to which 'else'.

Deprecated declaration *identifier* () – give arg types

In the prototype declaration of the named function, the argument's names and/or their types were not specified.

division by zero: *op*

Division, or remainder, by zero, will cause overflow.

Expected ')'; perhaps you tried to give too many names – *pragma ignored*

A ')' was expected but not found in a *pragma*; it may be that too many parameters have been given.

Expected integer as argument – *pragma ignored*

An integer argument was expected but not found in a *pragma*.

Expected string as argument – *pragma ignored*

The argument to the *IMS_linkage* *pragma* must be a string literal.

Expected string as fifth argument; *pragma ignored*

The fifth argument to the *IMS_descriptor* *pragma* must be a string literal.

Expected string as second argument – *pragma ignored*

The second argument to the *IMS_translate* *pragma* must be a string literal.

Expression generates poor code on this target ('dup' required)

An expression is being compiled for a transputer class of which only some members have the *dup* instruction. The compiler has decided that the expression could be compiled more efficiently using the *dup* instruction, but cannot do so because it is not present on all members of the class.

extern 'main' needs to be 'int' function

In a declaration of `main()`, the function should always be declared as type `int`.

extern *identifier* not declared in header

All objects must be declared before use. This message is only generated if specifically enabled by the 'FM' compiler option.

floating point constant overflow: *op*

Floating point overflow occurred during addition, subtraction, multiplication or division of two constants.

Floating-point generates poor code on this target

Floating-point code is being compiled for a transputer class of which only some members have instruction set additions to enhance floating-point performance. As these instructions are not present on all members of the class, the compiler cannot use them.

floating to integral conversion failed

Conversion (casting) from a floating point type to an integral type (such as `int`) failed.

formal parameter *identifier* not declared – 'int' assumed

A formal parameter has been listed in the parameter list of the function definition, but there is no entry for it in the declaration list; it is therefore assumed to be of type `int`.

Format requires *count* parameter(s), but *count1* given

A call to `printf` or `scanf` was made with the incorrect number of arguments. The control string indicated that *count* arguments are needed, but *count1* were provided. This warning is only generated if pragma `IMS_on` (`pc`) is active. The header file `stdio.h` includes this pragma.

Illegal format conversion '*%char*'

The character sequence '*%char*' is not a legitimate conversion character for `printf` or `scanf`. This warning is only generated if pragma `IMS_on` (`pc`) is active. The header file `stdio.h` includes this pragma.

Illegal language type *string*; replaced by *string*

The language type given for the interface descriptor-string is not a valid one, and has been overridden by a known type.

implicit cast (to type) overflow

Overflow occurred when casting an expression.

implicit narrowing cast: *op*

The result of an operation performed at higher precision is immediately, and implicitly, cast to lower precision, thus losing the extra precision: if the extra precision is not required, the operation ought to be performed at the lower precision.

If the narrowing cast is really required, the warning may be suppressed by writing the cast explicitly.

implicit return in non-void *identifier* ()

The function does not contain a `return` statement, even though it is defined to return a value.

Incomplete format string

The control string for use with `printf` or `scanf` is incomplete. This warning is only generated if pragma `IMS_on (pc)` is active. The header file `stdio.h` includes this pragma.

index value *number* is outside array bounds

The array subscript value *number* is larger than the maximum subscript allowed for the array, or smaller than the minimum subscript allowed for the array.

Integer too large to be represented – pragma ignored

An integer parameter to a pragma has been given with a value too large to be able to be dealt with by the compiler.

inventing 'extern int *identifier* ();'

No declaration exists for the function; it will be defined by default as `extern int`.

label *identifier* was defined but not used

The named label was set, but not used.

Linkage already set - pragma ignored

The `IMS_linkage` parameter has been specified more than once.

lower precision in wider context: *op*

The result of an operation performed at lower precision is immediately cast to a higher precision; it may be that the user was expecting the operation to be performed at the higher precision.

Missing comma in pragma argument list – pragma ignored

Multiple arguments to a pragma must be separated by commas.

Negative value given for vectorspace – pragma ignored

Vectorspace values in the `IMS_descriptor` pragma must be ≥ 0 .

Negative value given for workspace – pragma ignored

Workspace values in the `IMS_descriptor` pragma must be ≥ 0 .

No pragma name given in pragma directive – was this intended?

The compiler has detected a pragma directive which does not have a name. This is not illegal, however, it has no effect.

no side effect in void context: *identifier*

The value which has been returned by an expression is not being used e.g.

```
int a;  
a;
```

non-portable – not 1 char in '...'

The characters enclosed by single quotes represent more than one character. The compiler will read the first character only, for example, 'AB' will be read as 'A'.

Non-positive values for patch size are meaningless – pragma ignored

Patch size values must be > 0 .

non-value return in non-void function

A function which should return a value has terminated without using a return statement or with a return statement that has no arguments. The value received from the function by the calling routine is undefined.

odd unsigned comparison with 0 : *op*

$a \geq$ comparison of an unsigned integer with zero, or $a \leq$ comparison of zero with an unsigned integer, is always true.

omitting trailing '\0' for char [*count*]

The char array is fully occupied by characters and there is no room to append the string terminator (`\0`). *count* is the full length of the character array.

repeated definition of #define macro *identifier*

The named macro has been defined more than once. The definitions are identical.

Shift of type by *count* undefined in ANSI C

A shift of more than the number of bits in *type*, or less than zero was requested, undefined in ANSI C.

signed constant overflow: *op*

Overflow occurred when performing *op* upon signed, constant operands.

spurious {} around scalar initialiser

A scalar can take only one initializer, so there is no need to use braces as are required with aggregate types such as arrays.

static *identifier* declared but not used

The named static object was declared but not used.

struct has no named member

A structure has been declared without any members.

Too many assembler arguments**Too many compiler arguments**

There are too many options on the command line. The extra options are ignored.

Undefined macro *string* in *#if* – treated as 0

This error occurs when enumeration or undefined constants appear after the preprocessor *#if* directive. For example, if '*ab*' and '*cd*' are enumeration constants of the enumerated type '*abcd*', the statement *#if ab == cd* would generate this error.

union has no named member

A union has been declared without any members.

unnamed bit field initialised to 0

A static declaration of a structure or union containing an unnamed bit field, the compiler has initialized that field to zero.

Unrecognised *#pragma* (no '(')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised *#pragma* (no ')')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised *#pragma identifier*

identifier is not a pragma recognized by this compiler.

unsigned constant overflow: *op*

Overflow occurred when performing *op* upon unsigned, constant operands.

unused earlier static declaration of *identifier*

The static variable *identifier* has been defined before being declared. Generated only if the '*EH*' compiler option is specified.

use of *op* in condition context

Generated when the invalid operators '=' (assignment) or '~' (bit-not) are used in a condition statement.

This message is given for use of the assignment operator in condition context, e.g.

```
if (a = b)
```

as this is often due to mistyping the equality operator, i.e. the desired code is:

```
if (a == b)
```

If you really wish to perform the assignment in condition context, the warning message may be suppressed using the form:

```
if ((a = b) != 0)
```

wrong number of parameters to *function*

A function declared without a prototype was called with the wrong number of arguments. (An error is given if a function declared with a prototype is called with the wrong number of arguments.)

variable *identifier* declared but not used

The variable was declared, but not used anywhere in the program.

(possible error): >= *number* lines of macro arguments

There are a surprisingly large number of lines of arguments to a macro; this may indicate a syntax error.

1.7.6 Recoverable errors***#ident* is not in ANSI-C**

#ident is not a recognized preprocessor directive.

***##* first or last token in *#define* body**

The *##* preprocessor operator must be preceded by a preprocessor token, and succeeded by a preprocessor token.

***instruction* may not have a size specified**

An *__asm* pseudo-instruction may not be explicitly sized.

***','* (not *;'*) separates formal parameters**

A semicolon has been used to separate the formal parameters in a function definition (as in Pascal) instead of a comma.

***'register'* attribute for *identifier* ignored when address taken**

An attempt was made to take the address of a variable with 'register' storage class. The register attribute will be ignored allowing the address to be taken.

<int> op <pointer> treated as <int> op (int) <pointer>

The expression involving a integer and a pointer will result in the pointer being converted (cast) to an integer.

object identifier may not be function – assuming pointer

An attempt was made to use a function where it was not expected, typically when a function is included as a component within a structure.

op: cast between function and non-function object

The operation is performed upon two arguments, one of which is a function, and the other an object.

op : cast to non-equal type illegal

A structure or union has been cast into a structure or union of a different type. The cast is illegal and will be ignored.

op : illegal cast to type

An illegal cast has been attempted. The cast is illegal and will be ignored.

op : implicit cast of type to 'int'

A non-integer object has been used where an `int` was expected, for example, attempting to use a `double` as an argument to a `switch` statement (which requires an integer type).

op : implicit cast of non-0 int to pointer

Evaluation of the expression will result in the cast of an integer to a pointer.

op : implicit cast of pointer to 'int'

Evaluation of the expression will result in the cast of the pointer to an integer.

op : implicit cast of pointer to non-equal pointer

Evaluation of the expression will result in the cast of one pointer type to another.

op may not have whitespace in it

Two-character operators such as `+=` must not contain spaces.

<pointer> operator <int> treated as (int) <pointer> operator <int>

Evaluation of the expression will result in the cast of the pointer to an integer.

Ancient form of initialisation, use '='

A `}`, rather than `=`, was used to introduce an initializer, this is no longer legal C.

ANSI C does not support 'long float'

An object has been declared of type `long float`, this is illegal in ANSI C, which supports `float`, `double`, or `long double`.

Array of type illegal — assuming pointer

An array of functions or void objects has been declared. The compiler treats this as an array of pointers to functions or void objects.

Array [0] found

An empty array has been defined and will be set up instead as an array with one element.

assignment to 'const' object *identifier*

The expression contains an assignment to a constant. The assignment will be carried out.

const typedef *identifier* has const respecified

A typedef which is already qualified with `const`, has been qualified with `const`.

comparison *op* of pointer and int: literal 0 (for == and !=) is only legal case.

The specified operator was used to compare an object of type `int` and one of a type `pointer`. The only legal comparison of this type is between a pointer and 0 using either `==` or `!=`.

declaration with no effect

No name has been declared for the object. Specifying only the type of an object generates this error.

differing pointer types: *op*

The specified operator was used with pointers of different types.

differing redefinition of #define macro *identifier*

The named macro has been defined more than once. The definitions are not identical.

Digit 8 or 9 found in octal number

8 and 9 are meaningless in an octal number.

duplicate macro formal parameter: '*identifier*'

The function macro has two formal parameters with the same name.

duplicate member *identifier1* of *identifier2*

Two fields of structure or union *identifier2* have the name *identifier1*.

ellipsis (...) cannot be only parameter

A function declared to take a variable number of parameters must have at least one known parameter.

enumeration constant *identifier* too large to represent as 'int' – 0 assumed

The value of an enumeration constant has overflowed the range of ints.

extern *identifier* mismatches top-level declaration

An **extern** declaration of *identifier* within a function definition does not match an **extern** declaration of *identifier* at the top level.

expected *symbol1* or *symbol2* –inserted *symbol1* before *symbol3*

symbol1 or *symbol2* was expected before *symbol3*, but neither was found. *symbol1* is suggested as the most appropriate choice and the compiler has changed the code accordingly.

formal name missing in function definition

The type of a formal parameter has been omitted in a function definition.

function *identifier* may not be initialised –assuming function pointer

Initializers cannot be used in function declarations or definitions.

function prototype formal *identifier* needs type or class – 'int' assumed

The type of a formal parameter has been omitted in a function declaration and **int** has been assumed.

function returning *type* illegal — assuming pointer

The user has appeared to declare a function which returns a function or an array.

hex number cannot have exponent

A hex number ending in **e** may not be immediately followed by **+** or **-**; separate the number and the additive operator with white space.

illegal bit field *type* – 'int' assumed

Bit fields cannot be set within non integral variables. The compiler assumes an **int** instead.

illegal string escape '\char' – treated as *char*

The character following **** does not form part of a valid string escape. The compiler treats the sequence **\char** as *char*.

illegal [] member: *identifier*

An open array may not be a member of a structure or union.

junk at end of #*identifier* line – ignored

The text following the directive is invalid and will be ignored.

linkage disagreement for *identifier* – treated as *store class*

The storage class of a previously defined **static** or **extern** object or function disagrees with the current declaration. The object will be treated as though it is in storage class *store class*.

L'...' needs exactly 1 wide character

A wide character constant should contain exactly one wide character.

Missing newline before EOF – inserted

A blank line should have been inserted before the end-of-file character.

Missing type specification – 'int' assumed

A type specification is missing. The object will be assumed to be of type **int**.

more than 4 chars in character constant

More than 4 ASCII characters were used to represent a character constant. When using the single quote syntax for character constants a maximum number of 4 characters is permitted in order to accommodate the octal representation of a character. The first 4 characters will be used.

no chars in character constant "

No characters or character codes have been specified for the character constant. A NULL character is assumed.

no initializer list in braced initializer

There must be at least one entry in the initializer list of a braced initializer.

number illegally followed by letter

A numerical constant may not be followed immediately by a letter.

number missing in #line

There is no line number following the preprocessor **#line** directive.

objects that have been cast are not l-values

An object that has been cast in l-value context; ANSI has made this illegal.

Omitted type before formal declarator – 'int' assumed

No type was specified; type **int** will be assumed.

operand of # not macro formal parameter: '*identifier*'

The operand to the **#** preprocessor operator must be a formal parameter of the function macro containing it.

overlarge escape '*number1*' treated as '*number2*'

An octal number in an escape sequence is too large to be represented in the target architecture.

overlarge escape '\xnumber1' treated as '\xnumber2'

A hexadecimal number in an escape sequence is too large to be represented in the target architecture.

parentheses (. .) inserted around expression following *text*

Parentheses were expected after the specified text, for example, around a conditional expression such as an `if` statement.

prototype and old-style parameters mixed

It is illegal to mix new (prototype) and old-style parameter declarations.

return *expression* illegal for void function

A return statement with an expression was found within a `void` function. The return statement is ignored.

size of 'void' required – treated as 1

'void' was used as an argument to `sizeof`. The compiler assumes the size of void to be 1.

size of a [] array required, treated as [1]

The array is of unspecified size. In these circumstances `sizeof` return the size of the array type.

size of function required – treated as size of pointer

A function name was passed to the `sizeof` function. In these circumstances `sizeof` returns the size of the pointer to the function.

sizeof *bit field* illegal – sizeof(int) assumed

A bit field was passed to the `sizeof` function. In these circumstances `sizeof` casts the bit field to an integer and then returns its size.

Small (single precision) floating value converted to 0.0

The number is too small to represent in a single word (32 bit) floating point format, and has been rounded to 0.0.

Small floating point value converted to 0.0

The number is too small to represent in a double word (64 bit) floating point format, and has been rounded to 0.0.

Spurious `#elif` ignored

The `#elif` directive could not be matched with a corresponding `if` directive and has been ignored.

Spurious `#else` ignored

The `#else` directive could not be matched with a corresponding `if` directive and has been ignored.

Spurious `#endif` ignored

The `#endif` directive could not be matched with a corresponding `if` directive and has been ignored.

static function *identifier* not defined –treated as `extern`

A function was defined as `static` in the function prototype, but the compiler was unable to find the function definition. An `extern` function is assumed.

string initialiser longer than `char [count]`

A character array has been initialized with more characters than the array can accommodate. Since the compiler adds a terminating NULL character to strings, string initializers should always contain one less element than the array.

struct member *identifier* may not be function – assuming pointer

A structure member was declared of function type; the compiler treats this as pointer to function type.

struct tag *identifier* not defined

A structure has been referenced before being defined.

Translation unit contains no external declarations

A translation unit must contain at least one external declaration.

type or class needed (except in function definition) –‘int’ assumed

The type or storage class has been omitted from the function declaration.

Undeclared name, inventing ‘`extern int identifier`’

An undeclared identifier was encountered and will be given the storage class `extern`.

union member *identifier* may not be function – assuming pointer

A union member was declared of function type; the compiler treats this as pointer to function type.

union tag *identifier* not defined

A union has been referenced before being defined.

unprintable char *number* found - ignored

An unprintable character was found in the source text.

volatile typedef *identifier* has volatile respecified

A typedef which is already qualified with `volatile`, has been qualified with `volatile`.

wrong number of parameters to *function*

A function was called with the wrong number of arguments.

1.7.7 Serious errors***\space* and *\tab* are invalid string escapes**

White space (*'\space'* or *'\tab'*) was found within a string. All characters up to the first non-white space character are ignored; if the first non-white space character is a newline character, this will also be ignored.

***{}* must have 1 element to initialise scalar or auto**

When initializing a scalar quantity or *auto* variable only one initializer should be specified within the enclosing braces.

#error* encountered *string

The *#error* directive was found.

***#include* file *filename* wouldn't open**

The file *filename* could not be opened.

***op* : cast to non-equal type illegal**

A structure or union has been cast into a structure or union of a different type. The cast is illegal and will be ignored.

***op* : illegal cast of type to pointer**

A variable has been cast into a pointer type. The cast is illegal and will be ignored.

***op* : illegal cast to type**

An illegal cast has been attempted. The cast is illegal and will be ignored.

***context*: illegal use in pointer initialiser**

An object of type *auto*, or its address, cannot be initialized.

***(. . .)* must have exactly 3 dots**

An ellipsis must consist of three dots.

'break' not in loop or switch – ignored

A break statement was encountered outside the scope of a loop or switch statement. A break at this point is illegal and will be ignored.

'case' not in switch – ignored

A case prefix has been encountered outside the body of a switch statement. A case statement at this point is illegal and will be ignored.

'continue' not in loop – ignored

A **continue** statement has been encountered outside the body of a loop. A **continue** statement at this point is illegal and will be ignored.

'default' not in switch – ignored

A **default** prefix has been encountered outside the body of a **switch** statement. A **default** prefix at this point is illegal and will be ignored.

'goto' not followed by label – ignored

The text following a **goto** statement does not represent a label.

'void' values may not be arguments

Formal parameters in function definitions or declaration cannot be of type **void**.

'while' expected after 'do' – found *text*

The **while** statement is missing from a **do . . . while** construct. *text* marks the position.

'{' of function body expected – found *text*

The opening brace in the body of a function is missing.

'{' or <identifier> expected after *type*, but found *text*

The opening brace following a **struct**, **union** or **enum** is missing. *text* marks the position.

<asm-directive> expected but found a *text*

text indicates where the **__asm** directive was expected.

<command> expected but found a *text*

Statements such as **switch** or **if** should be followed by a command. *text* indicates where the command was expected.

<expression> expected but found *text*

text indicates where the expression was expected.

<identifier> expected but found *text* in 'enum' definition

The compiler was expecting to read an enumeration constant when it found *symbol*. This may be because there is a spurious comma at the end of a list of enumeration constants.

***function* has **pragma nolinek** specified, but accesses static data**

The specified function has been specified not to receive a static link (via **IMS_nolinek**), but attempts to use static data. It is only possible to use static data when a static link is available.

identifier is not a label - ldlabelfdiff ignored

The operands to the `ldlabelfdiff` pseudo-instruction must be labels.

instruction not followed by label - ignored

A load or store `__asm` instruction must have a constant or label operand.

store class variables may not be initialised

Some types of C variables, such as those declared as `extern`, cannot be initialized.

Array size count illegal – 1 assumed

Arrays cannot be larger than 0xfffff on a 32-bit target, or 65535 on a 16-bit target.

attempt to apply a non-function

A name not declared as a function has been used in a context where a function should be.

attempt to include struct/union identifier object/member within itself

A structure or union declaration may not contain a field of the structure or union type, or a field which references another field.

bit fields do not have addresses

Elements of type bit field in C structures cannot be addressed.

Bit size size illegal – 1 assumed

Bit sizes greater than 32 are set to 1.

Cannot call function (it requires a static link)

An attempt has been made to call the specified function which requires a static link, from a function which has been specified not to receive a static link (via `IMS_nolink`).

Cannot call through pointer (it requires a static link)

An attempt has been made to call a function through the specified pointer from a function which has been specified not to receive a static link (via `IMS_nolink`). All calls through function pointers are assumed to require a static link.

Cannot store to identifier

identifier is a built-in name, such as `_lsb` or `_params`, which cannot be assigned to.

char and wide (L"...") strings do not concatenate

A char string and a wide char string appear adjacently in the source text. Normally, adjacent strings in the source text are concatenated; however, this is not possible here, as they have different types.

Digit required after exponent marker

Exponents of floating point numbers must be followed by a numeric character. The numeric character may be preceded by '+' or '-'.

duplicate 'default' case ignored

The default prefix has already been specified for the switch construct. The original definition will be used.

duplicate definition of *identifier*

The named identifier has already been defined.

duplicate definition of *struct/union* tag *identifier*

The named structure or union identifier has already been used.

duplicate definition of label *identifier* – ignored

The specified identifier has already been used. The original definition will be used.

duplicate type specification of formal parameter *parameter*

The specified parameter has been listed more than once in the function's formal parameter list.

duplicate case constant: *constant*

The constant has been specified more than once in the same case statement.

EOF in comment

The end-of-file was detected inside a comment.

EOF in string

The end-of-file was detected within a string.

EOF in string escape

The end-of-file was detected within a string escape sequence.

EOF not newline after `#if` . . .

The end-of-file was found after the '`#if`' directive; a newline character was expected.

expected *symbol1* – inserted before *symbol2*

symbol1 was expected before *symbol2* and the compiler has changed the code accordingly. For example, in the code "`if (TRUE printf()) ;`" the compiler would expect to find ')' before '`printf`'.

Expected <identifier> after *operator* but found *text*

The specified operator must be followed by an identifier. This error may occur after the structure member operator '.' and the structure pointer operator '->'.

Expecting <declarator> or <type>, but found *text*

An identifier or type was expected at *text*. For example, the declaration `'typedef int *[3] test;'` generates this error.

Grossly over-long floating point number

There are too many digits in the floating point number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long hexadecimal constant

There are too many digits in the hexadecimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long number

There are too many digits in the decimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Hex digit needed after 0x or 0X

The hexadecimal specifier `0x` must be followed by a valid hexadecimal digit. The compiler assumes a zero digit.

Identifier (*name*) found in <abstract declarator> – ignored

An identifier should not be used in an abstract declarator. This error is generated, for example, if `sizeof(int *test[3]);` is used instead of the correct form `sizeof(int *[3]);`.

illegal bit field type *type* – 'int' assumed

Bit fields cannot be set within non integral variables. The compiler assumes an `int` instead.

**illegal character (*number* = '*char*') in source
illegal character (hex code *number*) in source**

An unexpected character was found in the source code. The ASCII code of the character (if printable), and the character itself, are given.

illegal in *context*: *error*

Illegal expressions such as those involving division by zero generate this error.

illegal in *expression*: non constant *identifier*

A constant is required in certain expressions, for example after a `case` prefix.

illegal indirection on (void *): ****

An attempt has been made to take the contents of the object pointed to by a pointer to void.

Illegal in l-value: *context*

An l-value was expected. For example, attempting to assign a value to a constant will generate this error.

Illegal in l-value: 'enum' constant *identifier*

Enumeration constants cannot be used as l-values in an expression.

Illegal in l-value: function or array *identifier*

Arrays and function declarators cannot be used as l-values. This error would be generated, for example, by attempting to assign a value to a function declarator.

Illegal in the context of an l-value: *op*

The operator *op* cannot appear in l-value context.

Illegal types for operands: *operator*

The operator has been used with an invalid type. For example, it is illegal to use the structure member operator '.' with a variable of type `int`.

Illegal 'void' member/object: *identifier*

An object or member of a structure or union cannot be declared as being of type `void`.

incomplete tentative declaration of *identifier*

The declaration of *identifier* has gone out of scope before the declaration has been completed.

Invalid command line option (*text*)

text is not a recognized command line option.

Invalid source file name (*filename*)

filename is not a valid source file name. (Source file names may not contain hyphens.)

I/O error writing *filename*

An error occurred when writing to the named file.

Junk after `#if` *expression*

The `#if` directive must be terminated by a newline character.

Junk after `#include` *filename*

The `#include` directive must be terminated by a newline character.

label *identifier* has not been set

A label has been referenced but not set. This message will be generated if `goto` is used with an undefined label.

ldlabdiff not followed by label - ignored

The operands to the `ldlabdiff` pseudo-instruction must be labels.

Misplaced 'else' ignored

An `else` statement was found where it was not expected. It will be ignored.

Misplaced '{' at top level – ignoring block

An opening brace was found at the top level of a program when it was not expected, for example when not used as part of a function or structure definition.

Misplaced preprocessor character *char*

A preprocessor directive character (`#` or `\`) was found where it was not expected.

Missing #endif at EOF

An `#endif` directive is missing. This error will not be generated until the last of the currently open files is about to be closed (ANSI standard does not require `#if` and `else` statements to match in included files).

Missing *char* in preprocessor command line

A 'quote' character is missing from a preprocessor command line. The missing character could be `'`, `<`, `>`, or `"`.

Missing ')' after *identifier* (... on line *number*

A closing parenthesis is missing from the macro which will be substituted at line *number*.

Missing '*'* or ')' after #define *identifier* (...

The list of parameters in a macro definition is either incomplete or has not been correctly terminated by a closing parenthesis.

Missing < or " after #include

The opening 'quote' character which introduces the filename is missing.

Missing hex digit(s) after \x

The hexadecimal introducer sequence `\x` was found, but no hexadecimal digit was specified. The compiler assumes that the letter `x` was intended.

Missing identifier after #define

The definition is empty. `#define` must be followed by an identifier.

Missing identifier after #ifdef

`#ifdef` must be followed by an identifier.

Missing identifier after #undef

`#undef` must be followed by an identifier.

Missing include directory name

The **J** command line option must be followed by a directory name.

Missing map file name

The **P** command line option must be followed by a map file name.

Missing object file name

The **O** command line option must be followed by an object file name.

Missing parameter name in #define *identifier* (. . .

A parameter is missing from the specified macro definition. This error would be generated by a definition of the form **#define test(arg,).**

Newline or end of file within string

A newline or end-of-file character was encountered within a string.

No '*'* after #if defined(. . .

The closing parenthesis is missing from the directive.

No file name given

No source file was specified on the command line.

No identifier after #if defined

#if defined must be followed by an identifier.

Non-formal *identifier* in parameter-type-specifier

The parameter *identifier* was included in the declarator list of a function, but not in the parameter list. For example, a definition such as **int foo () int x; { }** would generate this error.

non-static address *identifier* in pointer initialiser

Pointers cannot be initialized with the address of an object of type **auto**.

Number *number* too large for 32-bit implementation

The specified number is too large to be represented in 32 bits.

objects of type 'void' can not be initialised

Initializing objects of type **void** is illegal.

only const and volatile can qualify a pointer: found *type*

The only type qualifiers of a pointer are **const** and **volatile** but *type* was found instead.

Operand *number* to instruction is larger than a word

The arguments to an **__asm** load or store pseudo-instruction must fit in a machine word.

Operand *number* to *instruction* is not word-sized

The arguments to an `__asm` store pseudo-instruction must fit exactly in a machine word.

Operand to *instruction* must be a constant or local variable

An illegal operand has been given to an `__asm` `ldl` or `stl` instruction.

Operand to *instruction* is larger than a word

The operand to a primary instruction inside `__asm` must fit in a machine word.

Out of memory**Out of store (for error buffer)****Out of store (in `cc_alloc`)**

The compiler ran out of available memory.

Overlarge (single precision) floating point value found

The number is too large to represent in single word (32 bit) floating point format.

Overlarge floating point value found

The number is too large to represent in double-word (64 bit) floating point format.

quote (*char*) inserted before newline

The specified quote character was found before a newline character. This may indicate a spurious character or a missing closing quote.

re-using *struct/union* tag *identifier* as *union/struct* tag

The named identifier has been used to identify two different types of object.

size of expression unknown: treated as 0

The size of a structure or union is required, but the structure or union has not been completely declared.

size of *struct identifier* needed but not yet defined**size of *union identifier* needed but not yet defined**

The size of the structure/union has not yet been defined. This error can occur when an undefined structure/union is used as an argument to the `sizeof` function and when an undefined structure/union is used in the declaration of a variable. In the second case the error occurs because the compiler attempts to determine the size of the structure/union for memory allocation purposes.

storage class *store class* incompatible with *store class* – ignored

Two incompatible storage classes have been used in a declaration. For example, `extern static foo;` generates this error because `extern` and `static` are incompatible types.

storage class *store* class not permitted in context *context* – ignored

The specified storage class is not permitted in the context in which it has been used. This error would be generated, for example, if storage class *auto* were to be used at the top level.

struct *identifier* has no *identifier* field

The structure contains no field of that name.

struct *identifier* must be defined for (static) variable declaration

An undefined structure has been used in a variable declaration.

struct *identifier* not yet defined –cannot be selected from

A reference was made to an undefined structure.

Too few operands for *instruction*

A load or store *__asm* pseudo-instruction has too few arguments.

Too few arguments to macro *identifier*(. . . on line *number*

There are too few arguments to the macro which will be substituted at line *number*.

Too many operands for *instruction*

A load or store *__asm* pseudo-instruction has too many arguments.

Too many arguments to macro *identifier*(. . . on line *number*

There are too many arguments to the macro which will be substituted at line *number*.

Too many errors

After 100 Serious errors, the compilation aborts.

too many initialisers in {} for aggregate

An aggregate type, for example an array, has been initialized with more values than can be accommodated.

type *type1* inconsistent with *type2*

Two incompatible type identifiers are being used in the declaration of a single object. For example, the declaration `double int x;` would generate this error.

type disagreement for *identifier*

The specified identifier has already been assigned a different type.

typedef name *type* used in expression context

A type definition has been used in an expression.

type qualifier *type qualifier* not allowed to qualify *type qualifier* **type**

'const' may not be repeated in the qualifying list of a type, and similarly for 'volatile'.

undefined struct/union *identifier1* member/object: *identifier2*

The structure or union is, at present, undefined.

Uninitialised static [] arrays illegal

Static arrays of unspecified size must be initialized.

union *identifier* has no *identifier* field

The union contains no field of that name.

union *identifier* must be defined for (static) variable declaration

An undefined union has been used in a variable declaration.

union *identifier* not yet defined –cannot be selected from

A reference was made to an undefined union.

Unknown directive: *#identifier*

identifier is not a valid preprocessor directive. Check spelling and/or syntax.

unknown instruction *instruction*

instruction is not a defined transputer instruction.

zero width named bit field – 1 assumed

Named bit fields must be at least one bit wide.

2 `icconf` - configurer

This chapter describes the configurer tool `icconf` that configures code for transputer networks. It describes the command line syntax and explains how the tool generates a configuration data file from a configuration description for input to the code collector tool. The chapter ends with a list of configurer diagnostics and error messages.

2.1 Introduction

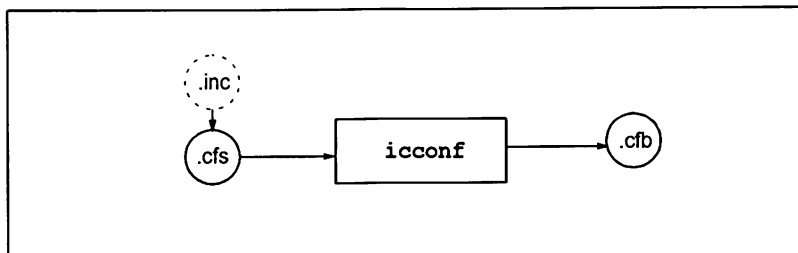
The configurer takes a *configuration description* created using the transputer configuration language and produces a *configuration data file* which `icollect` uses to generate bootable code for a transputer network.

A configuration description describes how code is to be run on a network of transputers. It consists of separate definitions of the software and hardware networks, and a mapping description which defines how the software will be placed on the processor network. Using this description the configurer allocates code to particular processors and performs wide ranging consistency checks on the mapping of software to hardware.

`icconf` enables any topology of software network to be placed on any topology of hardware network. There are no restrictions on how many communication channels may be allocated to a single inter-processor link. Where possible channels should be left unplaced by the user, so that `icconf` can implement the 'best' route through the network. Processes must be allocated to specific processors and any channels going to edges must be placed on the specific edge links.

Code to be run on separate processors must be linked code. Linked units that are to be run on the same transputer must be compiled for the same or a compatible transputer type.

The operation of the configurer tool in terms of the standard toolset file extensions is illustrated below.



2.2 Configuration language implementation

The configuration language supported by `icconf` has a number of implementation characteristics of which the programmer should be aware. These are briefly listed below; details can be found in section B.2 of the *ANSI C Toolset User Guide*.

- Array subscript ranges are machine word-length dependent.
- Source lines must not exceed 1024 characters. Leading and following white space is ignored.
- The number of dimensions for identifiers and array constants must not exceed 64.

2.3 Running the configurer

The configurer takes as input a configuration description file and produces a configuration data file for input to the collector tool.

To run the configurer use the following command line:

► `icconf filename { options }`

where: *filename* is the configuration description file. The filename is interpreted as given and no file extension is assumed.

options is a list of one or more options from table 2.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Example of use:

UNIX based toolsets:

```
icc hello
ilink hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS/VMS based toolsets:

```
icc hello
ilink hello.tco /f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

Option	Description
C	Checks the configuration description only. No configuration data file is generated.
G	<p>This option is used when postmortem or interactively debugging and disables any ordering of process memory segments in the configuration code by the <code>order</code> and <code>location</code> process attributes.</p> <p>The G option significantly modifies the runtime behavior of the configured program because virtual though routing is used for all channel communication between processors. This results in a memory overhead and reduction in performance of communications.</p> <p>This option cannot be used with the GA, GP, RA or RO options.</p>
GA	Generates a configuration which can be debugged using the <i>Advanced Toolset</i> debugger. This option has the same side effects as the G option except that the <code>order</code> and <code>location</code> attributes are not disabled. This option cannot be used with the G or GP options.
GP	<p>This option is used when postmortem debugging and disables any ordering of process memory segments in the configuration code by the <code>order</code> and <code>location</code> process attributes.</p> <p>The runtime behavior of the application will be little different to the default behavior i.e. when no options are specified. Virtual routing is enabled and may be used. This option <i>may</i> be used with the RA option but <i>not</i> with the G, GA or RO options.</p>
I	Displays extra information as the tool runs.
NV	Generates a configuration without virtual routing.
O filename	Specifies an output filename. If no output file is specified the configuration data file is given the base name of the input file and the <code>.cfb</code> extension is added.
P procname	Specifies the name of the root processor when configuring for EPROMs. <i>procname</i> must not be an element from an array of processors.
PRE	Generates a configuration which can be profiled using the <i>Advanced Toolset</i> network execution profiler. This option has the same side effects as the GA option. Note: this option cannot be used with the GA or PRU options.
PRU	Generates a configuration which can be profiled using the <i>Advanced Toolset</i> network utilization profiler. This option has the same side effects as the GA option. Note: this option cannot be used with the GA or PRE options.

Table 2.1 icconf command line options, page 1 of 2

Option	Description
RA	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor and all other processors are loaded into RAM to execute.
RO	Creates a file suitable for a boot-from-ROM application in which the user and system processes for the root processor execute in ROM and for all other processors the user and system processes are loaded into RAM to execute.
RS <i>romsize</i>	Specifies the size of ROM on the root processor. Only valid when used with the 'RA' or 'RO' options. <i>romsize</i> is specified in decimal format and can be followed by 'K' or 'M' to indicate kilobytes or megabytes.
W	Disables configurer messages of severity <i>Warning</i> .
WP	Generates additional pedantic <i>Warning</i> messages.

Table 2.1 continued – Standard `icconf` compiler options

2.3.1 Default command line

Default command line parameters can be defined on the system in the `ICCONFARG` environment variable. Parameters must be specified using the syntax required by the configurer command line.

2.3.2 Virtual routing processes

The configurer will automatically add virtual routing processes if they are required. If virtual routing is not required the virtual router can be disabled by using the 'NV' command line option. **Note:** the use of this option also has an affect on the value of `LoadStart`, see section 2.3.11.

Chapter 6 of the ANSI C Toolset User Guide gives further information about virtual routing.

2.3.3 Support for the Advanced Toolset

The 'GA', 'PRE' and 'PRU' command line options support the use of the *Advanced Toolset's* debugging and profiling tools. These options have no affect within the scope of the *ANSI C Toolset* and should not be used.

2.3.4 Boot from ROM options

The boot-from-ROM options 'RO' and 'RA' indicate that the program is to be collected for loading into EPROM and select the execution mode (from ROM or RAM) for the root transputer code. The 'RS' option enables the size of ROM on the root processor to be specified.

2.3.5 Mixed language programming

When the program includes a mixture of C and OCCAM modules the configurer will perform some extra checks to ensure continuity exists. If the command line options used on `icconf` indicate that the program is to be interactively debugged or virtual routing is enabled then the configurer checks that interactive debugging is enabled in the OCCAM modules. If it is not, a warning will be issued.

2.3.6 Configurer library file

The configurer reads a special library file which contains the system startup processes for the different transputer types. The file is called `sysproc.lib` and is searched for on the directory specified by `ISEARCH`. This is normally the toolset `libs` directory, in which the file was originally installed.

A further file `sysvlink.lib` contains the virtual routing system processes placed by the configurer when virtual through-routing is required.

2.3.7 Standard include files

A number of standard include files are supplied to assist with configuration. All include files carry the `.inc` extension.

Defaults file `setconf.inc`

Configurer defaults are defined in the file `setconf.inc`. This file is automatically included at startup and does not need to be referenced by an `#include` statement.

`setconf.inc` contains a number of boolean constants, definitions of process and processor base types, and predefined INMOS processor types. `setconf.inc` is supplied on the `libs` installation directory.

Other include files

Two other include files are provided on the `libs` directory. These provide definitions of processor and memory combinations for INMOS *iq* systems products.

<code>trams.inc</code>	Processor type definitions for INMOS <i>iq</i> systems TRANSputer Modules (TRAMs).
<code>boards.inc</code>	Processor type definitions for INMOS <i>iq</i> systems transputer evaluation boards.

These two files are *not* automatically referenced by the configurer and need to be included in the normal way.

INMOS *iq* systems products are available separately through your local distributor.

2.3.8 Configuration description examples

A series of example configuration descriptions are supplied in the `icconf` examples subdirectory. These include configurations for specific network topologies such as rings, grids, trees, and pipelines.

Further simple configurations are provided in the `simple` examples subdirectory.

2.3.9 Search paths

If a directory path is not specified the configurer uses the standard toolset search mechanism for locating input files, include files, and system library files. Briefly, the current directory is searched first, followed by the directories specified by `ISEARCH` (if defined on the system). For details see appendix A.

2.3.10 Default memory map

By default the configurer maps code into memory in the following order beginning at `LoadStart`: stack; code; vector space; static; heap and system data. The memory segments are contiguous. The upper limit of the memory available to the configurer is defined in the configuration description file (`.cfs`), by the `memory` attribute specified for the processor node. The default memory map is illustrated in Figure 2.1. **Note:** vector space is only required if Occam modules are present in mixed language programs.

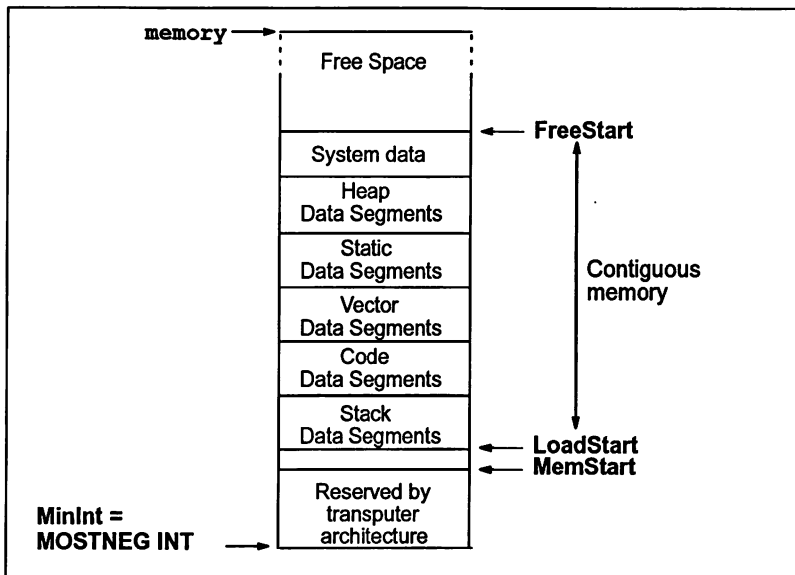


Figure 2.1 `icconf` default memory map

The first 2 or 4 Kbytes of memory above `MOSTNEG INT` is implemented as on-chip RAM, and includes a few words which are reserved by the transputer hardware for the implementation of links and other hardware registers. `LoadStart` is either just above or coincident with `MemStart`, see below. `FreeStart` is the start of unused memory.

2.3.11 LoadStart

The position of **LoadStart** for a processor varies depending on the use of **icconf** command line options and the **reserved** processor attribute, optionally specified within a configuration description.

When the **reserved** processor attribute is specified, **LoadStart** is defined to be the memory location obtained by adding the value of **reserved** to **MOSTNEG INT**.

When the **reserved** processor attribute is *not* specified, **LoadStart** is coincident with or just above **MemStart**:

- **LoadStart** = (**MOSTNEG INT** + 40 words) when *no* command line options are used i.e. virtual routing support is enabled.
- **LoadStart** = (**MemStart** + 6 words) when the '**NV**' command line option is specified, disabling virtual routing, but debugging or profiling is enabled either for **idebug** or the *Advanced Toolset*.
- **LoadStart** = **MemStart** when the '**NV**' command line option is specified but neither **idebug** or the *Advanced Toolset* debugger or profiler are used.

The value of **LoadStart** can be checked once the application has been collected, by generating and examining the collector map file, see chapter 3.

2.3.12 System processes

By default, the system startup processes' code and data are placed into user process data areas e.g. stack, code, vector and so on. These system processes do not interfere with the user's data because they complete their task before the space is needed by the user's code. **Note:** these system processes do not include the virtual routing processes placed by the configurer.

2.4 Configurer messages

Errors in the configuration source produce error messages in standard toolset format. Details of the format can be found in section A.7.

Messages are generated at *Information*, *Warning*, *Error*, *Serious* and *Fatal* severities. Most messages are generated at *Error* severity. The configurer aborts after 400 source file errors.

In the following lists messages are grouped by severity and listed in alphabetical order.

2.4.1 Information

The following messages are generated at severity level *Information*. They are enabled by the **I** command line option.

isolated root processor 'name'

If multiple routing sub-networks are required then the named processor has been selected as the root processor within its sub-network because it is the only processor in its sub-network. This processor will have no connection to any other processors via the basic spanning trees.

no through routing on 'name'

The named processor has been prevented from being used for through-routing channels by assigning its **routecost** attribute to be **INFINITE_COST** or greater.

placed channel 'name' onto link 'name'

The named channel has been placed automatically by the configurer onto the named link.

placed edge 'name' onto edge 'name'

The named input (or output) edge has been placed automatically by the configurer onto the named hardware edge.

placing *string* on 'name'

The named processor has been placed with a software through-routing kernel process. *string* is the module that has been placed.

selected root processor 'name'

The named processor has been selected as the root processor of the spanning tree derived to provide a basic route for all routed channels. If the application requires routing as multiple sub-networks then each sub-network will generate this message.

'name' I *string1* O *string2* Routers *count1* DeMuxes *count2* Muxes *count3*

The named processor requires virtual link support processes. *count1*, *count2*, and *count3* specify the number of through-routing, demultiplexing, and multiplexing modules. *string1* and *string2* describe the virtual links implemented by each input and output link of the processor.

These strings consist of pairs of characters; where the first is a digit and specifies the link number and the second, which can be **L**, **R**, or **x**, specifies if local data, local and through-routed data, or just through-routed data is carried by that link.

2.4.2 Warnings

The following messages are generated at severity level *Warning*.

attribute '*name*' definition ignored

This message can only occur in mixed language programs incorporating OCCAM modules. The named **stacksize** or **heapsize** attribute has been assigned a value that has been ignored.

attribute '*name*' has been reassigned

Named attribute has been reassigned.

attribute '*name*' undefined

Named attribute has not been assigned a value.

channel '*name*' unconnected and unplaced

Named channel has not been connected or placed.

connector '*name*' unused

Named connector has not been used in a **connect** statement.

could not through route between '*name1*' and '*name2*'

This message is output when the basic routing algorithm within the configurer cannot find a viable route between a pair of processors. This may explain subsequent errors output concerning channels running between these two processors. *name1* and *name2* identifies the processors that cannot be connected.

edge '*name*' unconnected and unplaced

The named input (or output) edge has not been connected or placed.

edge '*name*' unconnected

The named edge has not been connected.

exceeded LinkQuota on processor '*name*'; *count1* Inputs *count2* Outputs

This warning is output if the requested **linkquota** on any processor has been exceeded by the configurer. *name* identifies the processor concerned while *count1* and *count2* indicates the number of input and output links that are required.

ignored placement of *direction* channel on *link*

A channel connection placed onto the named link has been ignored. This is generated if channel connections between processors have been placed and interactive debugging has been specified. (The interactive debugger requires all channel connections between processors to be implemented as virtual channels.) *direction* can be either input or output.

illegal value for attribute '*name*' when profiling

The named attribute has been assigned an illegal value. This is generated if a process has been specified to start in high priority and the `PRE` or `PRU` options have also been specified.

insufficient memory size for attribute '*name*', *value*

The memory size specified by the named attribute is insufficient for the type of processor. This will be generated if the `memory` or `reserved` processor attributes have been assigned a memory size that is less than `LoadStart`. *value* is the memory size.

link '*name*' unconnected

Named link has not been connected.

nested comment statements, *value*

One or more nested comments have been found by the configurer. *value* is the number of nested comments found.

overflow in hexadecimal escape character

A numerical overflow has occurred during the evaluation of a hexadecimal escape character whose range is from 0 to 255.

overflow in octal escape character

A numerical overflow has occurred during the evaluation of an octal escape character whose range is from 0 to 255.

process location attributes ignored

The values assigned to the `location` process attributes have been ignored. These attributes are ignored if either the `G` or `GP` options have been specified.

process order attributes ignored

The values assigned to the `order` process attributes have been ignored. These attributes are ignored if the `G` or `GP` options have been specified.

processor 'name' unconnected

Named processor has not been connected to the network.

processor 'name' unused

Named process has been connected to the network but has had no user processes placed onto it.

process 'name' using direct channel input/output

The named process has been compiled to use direct instructions instead of indirect functions for channel i/o and because the process may also execute under the interactive debugger or with the virtual link support processes an incompatibility may arise.

process 'name' using indirect channel input/output

The named process has been compiled to use indirect functions instead of direct instructions for channel i/o and because the process will not be executed under the interactive debugger or with the virtual link support processes the use of functions for channel i/o is unnecessary.

unable to debug 'name' as not accessible from root

The named process cannot be debugged by the advanced toolset debugger because there exists no route, either physical or virtual, between the processor on which the process is placed and root processor (which is connected to the host).

using single hop software virtual links

This is output every time the the configurer adds any run-time multiplexing software to the user's program to support virtual channels.

using through routed software virtual links

This is output every time the the configurer adds any run-time multiplexing and through-routing software to the users program to support virtual channels.

string for process 'name' overlaps memory registers

A memory segment of the named process overlaps the hardware registers, which are the memory locations below **MemStart**, of the processor that the process has been placed on. *string* is the memory segment name which can be code, heap, stack, static or vector.

string of 'name' overlaps string of 'name'

A memory segment of the first named process overlaps a memory segment of the second named process. *string* is the memory segment name which can be code, heap, stack, static or vector.

2.4.3 Errors

The following messages are generated at severity level *Error*. Most configurer diagnostics are generated at this level.

attribute '*name*' cannot be reassigned

Named attribute cannot be reassigned. This can only occur with the **element** and **type** attributes of nodes and processors.

attribute '*name*' multiply defined in '*name*'

Named attribute has been declared within the **interface** attribute and its name clashes with a previously declared attribute or its name clashes with the name of a predefined attribute for the named process.

attribute '*name*' undefined in '*name*'

Named attribute is an undefined attribute of the named symbol.

attribute '*name*' undefined

Named attribute has not been assigned a value which is required.

automatic Tree Router provoked Deadlock II
automatic Tree Router provoked Deadlock

A potential communications deadlock has been detected in the routing network constructed by the virtual through-routing algorithm where a cycle of links that may through-route to each other has been created. This is an internal consistency error and should never be generated.

bad channel placement between '*name*' and '*name*'

A channel placement between the named processors specifies link numbers which do not correspond to each end of the same link connection. This is an internal consistency error and should never be generated.

cannot route channel between *name1* and *name2*

This message indicates a failure to complete auto-routing. The message is output each time a channel cannot be implemented either because it is too long (> 24 hops) or no viable route exists to support it. The *name1* and *name2* strings give the source and destination processors between which the data or acknowledge path of a channel cannot be routed.

channel '*name*' connected and unplaced

Named channel has been connected to an input (or output) edge and has not been placed onto a link.

channel 'name' multiply connected

Named channel has been used more than once in a `connect` statement.

channel 'name' multiply placed

Named channel has been used more than once in a `place` statement.

channel 'name' unconnected and placed

Named channel has been placed and has not been connected.

connect 'name' to 'name' illegal, both channels

An illegal `connect` statement has been specified where both named elements are channels and they communicate in the same direction.

connect 'name' to 'name' illegal, both edges

Connect statement is illegal because the named elements are both edges.

connect 'name' to 'name' illegal, channel/edge

An illegal `connect` statement has been specified where the first named element is a channel and the second named element is an input (or output) edge and they communicate in different directions.

connect 'name' to 'name' illegal, edge/channel

An illegal `connect` statement has been specified where the first named element is an input (or output) edge and the second named element is a channel and they communicate in different directions.

connector 'name' multiply placed

Named connector has been used more than once in a `place` statement.

connector 'name' multiply used

Named connector has been used more than once in a `connect` statement.

constant dimension sizes inconsistent, *value*

A constant array has been defined which has inconsistent dimension sizes for some of its elements. *value* is the number of the incorrect dimension, counting from zero.

constant dimensions incompatible with 'name'

Named symbol has been assigned a constant value whose dimensions are incompatible with those of the symbol.

constant element types not equal, *type*

A constant array has been defined where some or all of its elements have non-equal types. *type* is the expected type for each element in the array.

constant type incompatible with '*name*', *type*

Named symbol has been assigned a constant value whose type is incompatible with that of the symbol. *type* is the expected type for the constant.

edge '*name*' connected and unplaced

The named input (or output) edge has been connected to a channel and has not been placed onto a hardware edge.

edge '*name*' multiply connected

The named edge has been specified more than once in a `connect` statement.

edge '*name*' multiply placed

The named edge has been specified more than once in a `place` statement.

edge '*name*' unconnected and placed

The named edge has been placed and has not been connected.

element '*name*' in connection undefined

The named element has been used in a `connect` statement and is undefined. This is only generated from process and processor types.

element '*name*' in placement undefined

The named element has been used in a `place` statement and is undefined. This is only generated from process and processor types.

element '*name*' not completely subscripted

Named symbol has been defined as an array and has not been completely subscripted.

host edge '*name*' undefined

When configuring to boot from link, the named host edge has not been declared in the configuration source. This error can only be caused if the standard include file `setconf.inc` has been altered.

illegal # directive type, found 'string'

An illegal identifier for a #directive has been specified. *string* is the illegal directive identifier.

illegal 16 bit RAM + ROM memory size for 'name' , value

The named processor is a 16 bit processor which has been specified RAM and ROM memory sizes whose total is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit RAM memory size for attribute 'name' , value

The named processor is a 16 bit processor which has been specified a RAM memory size which is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit ROM memory size for 'name' , value

The named processor is a 16 bit processor which has been specified a ROM memory size which is illegal for 16 bit processors. *value* is the illegal memory size.

illegal 16 bit address for attribute 'name'

The named attribute, which is a sub-attribute of the `location` attribute for a process, has been assigned an address which is illegal for 16 bit processors. *value* is the illegal address.

illegal assignment for attribute 'name'

The named attribute has been specified in an attribute modification statement and is not of arithmetic type.

illegal definition of attribute 'name' for PRI PAR process

The named attribute, which is always the `priority` attribute of a process, has been assigned to `HIGH` when the code for the process has been specified to also execute at high priority.

illegal definition of attribute 'name' when executing from ROM

The named attribute, which is always the `code` attribute of the `location` attribute for a process, has been assigned an address when the code for the process is to execute from ROM.

illegal dimension size, value

A dimension size not greater than zero has been specified. *value* is the dimension number with the illegal dimension size.

illegal escape character sequence, *char*

An illegal escape character sequence has been specified. *char* is the illegal escape character.

illegal format character constant, *char*

An illegal format character constant has been specified. *char* is the unexpected character found in the character constant.

illegal format hexadecimal constant, *char*

An illegal format hexadecimal constant has been specified. *char* is the unexpected character found in the hexadecimal constant.

illegal memory size for attribute '*name*', *value*

The named attribute, which is always the **reserved** attribute of a processor, has been assigned a memory size which is greater than the size assigned to the **memory** attribute of the processor. *value* is the illegal memory size.

illegal number of dimensions for '*name*'

The named symbol has been declared as an array whereas it should have been declared as a scalar. This is generated for the host edge (when booting from link) or for the root processor (when booting from ROM).

illegal number of dimensions, *value*

Number of dimensions for a symbol or constant exceeds the maximum number of dimensions allowed by the configurer. *value* is the maximum number of dimensions allowed.

illegal number of subscripts for '*name*', *value*

Number of subscripts specified for the named symbol exceeds the number the symbol requires. *value* is the maximum number of subscripts allowed.

illegal number of subscripts for constant, *value*

Number of subscripts specified for a constant exceeds the number the constant requires. *value* is the maximum number of subscripts allowed.

illegal operation for attribute '*name*'

The named attribute has been used inappropriately in an attribute modification statement.

illegal source file character, *value*

An unexpected character has been found in the source file. *value* is the ASCII value for the illegal character.

illegal subscript value, *value*

A subscript value of less than zero or greater than the dimension size has been specified. *value* is the number of the dimension with the illegal subscript value.

illegal token for expression, found *token*

An unexpected token has been found at the start of an expression. *token* is the unexpected token.

illegal token for statement, found *token*

An unexpected token has been found at the start of a statement. *token* is the unexpected token.

illegal type for '*name*' in USE statement, *type*

Named symbol has been specified in a **use** statement and is not a process or a process type. *type* is the type of the symbol.

illegal type for '*name*' in connection, *type*

Named symbol has been specified in a **connect** statement and is not a channel, edge, link or connector. *type* is the type of the symbol.

illegal type for '*name*' in definition, *type*

Named symbol has been specified in a node definition statement and is not a node type. *type* is the type of the symbol.

illegal type for '*name*' in expression, *type*

Named symbol has been specified in an expression and is not a constant value. *type* is the type of the symbol.

illegal type for '*name*' in modification, *type*

Named symbol has been specified in an attribute modification statement and is not a node. *type* is the type of the symbol.

illegal type for '*name*' in placement, *type*

Named symbol has been specified in a **place** statement and is not a process, processor, edge, channel, link or connector. *type* is the type of the symbol.

illegal type for 'name' , type

Named symbol does not have the type expected by the configurer. This will only occur if the name specified using the `P` option is not a processor (when booting from ROM) or if the host edge `host` is in fact not an edge (when booting from link). *type* is the type of the symbol.

illegal type for IF statement condition, type

The condition value for an `if` statement is not of integral type. *type* is the type of the condition value.

illegal type for arithmetic operator operator, type

The operand of an arithmetic unary operator is not of arithmetic type. *type* is the type of the operand and *operator* is the arithmetic operator.

illegal type for boolean operator operator, type

The operand of a boolean binary operator is not of integral type. *type* is the type of the operand and *operator* is the boolean operator.

illegal type for condition operator operator, type

The condition value for a conditional ternary operator is not of integral type. *type* is the type of the condition value and *operator* is the conditional operator.

illegal type for connector 'name' in placement, type

Named symbol is a connector defining a connection and has been used in the incorrect position in a `place` statement. *type* is the type of connection defined by the symbol.

illegal type for dimension size, type

The type of a dimension size value is not of integral type. *type* is the type of the dimension size value.

illegal type for integral operator operator, type

The operand of an integral unary operator is not of integral type. *type* is the type of the operand and *operator* is the integral operator.

illegal type for subscript value, type

The type of a subscript value is not of integral type. *type* is the type of the subscript value.

illegal type for value in REP statement, *type*

The base or limit value for a replicator statement is not of integral type. *type* is the type of the base or limit value.

illegal types for arithmetic operator *operator*, *type1* and *type2*

The operands of an arithmetic binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the arithmetic operator.

illegal types for equality operator *operator*, *type1* and *type2*

The operands of an equality binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the equality operator.

illegal types for integral operator *operator*, *type1* and *type2*

The operands of an integral binary operator are not both of integral type. *type1* and *type2* are the types of the operands and *operator* is the integral operator.

illegal use of constant for element

A constant value has been used as an element.

illegal use of subfield operator for '*name*'

The named symbol, which has no accessible attributes, has been used with the subfield operator.

illegal use of subfield operator for constant

A constant value has been accessed using the subfield operator.

illegal value for attribute '*name*'

Named attribute has been given a value which is inconsistent with the type of the attribute and its semantic meaning.

incompatible interface, '*name*' has different type, *type*

incompatible interface, '*name*' has too few parameters

incompatible interface, '*name*' has too many parameters

incompatible interface, '*name*' has unequal dimensions

These messages can only be generated in mixed language programs incorporating OCCAM modules. The named symbol is an OCCAM process and the `interface` defined for the process mismatches the formal parameter list defined in the object file associated with the process in a `use` statement.

insufficient RAM memory for 'name' , value bytes amiss

Named processor's total RAM memory size is insufficient for the number of processes placed on the processor (which includes their data requirements). *value* is the number of extra bytes needed to accommodate all the processes on the processor.

insufficient ROM memory for 'name' , value bytes amiss

Named processor is the root processor in a boot from ROM system and its total ROM memory size is insufficient for the number of processes placed on the processor. *value* is the number of extra bytes needed to accommodate all the processes on the processor.

link 'name' multiply connected

Named link has been used more than once in a `connect` statement.

link 'name' multiply placed

Named link has been used more than once in a `place` statement.

links 'name' and 'name' unconnected and placed

Named links are not connected to each other and have each been placed with channels which are connected to each other.

missing (for SIZE operator, found *token*

The `size` operator has been found and an opening parenthesis was expected to be found after the keyword `size`, instead of which the token *token* was found.

missing) for SIZE operator, found *token*

The `size` operator has been found and a closing parenthesis was expected to be found after the operand to the operator, instead of which the token *token* was found.

missing) for attribute list, found *token*

An attribute list has been found and a closing parenthesis was expected to terminate the list, instead of which the token *token* was found.

missing) for cast operator, found *token*

A cast operator has been found and a closing parenthesis was expected to be found after the type identifier, instead of which the token *token* was found.

missing) for expression, found *token*

A parenthesized expression has been found and a closing parenthesis was expected to be found after the sub-expression, instead of which the token *token* was found.

missing , or TO for CONNECT statement, found *token*

A connect statement has been found and a comma or the keyword *to* were expected to be found, instead of which the token *token* was found.

missing : for conditional operator, found *token*

A conditional operator has been found and a colon was expected to be found after the first sub-expression, instead of which the token *token* was found.

missing ; for statement, found *token*

A statement has been found which expects a semicolon to terminate it, instead of which the token *token* was found.

missing = for REP statement, found *token*

A replicator statement has been found and an equals was expected to be found after the replicator identifier, instead of which the token *token* was found.

missing = or (for attribute, found *token*

An attribute definition has been found and an equals or opening parenthesis were expected to be found after the attribute identifier, instead of which the token *token* was found.

missing FOR for USE statement, found *token*

A use statement has been found and the keyword *for* was expected to be found, instead of which the token *token* was found.

missing ON for PLACE statement, found *token*

A place statement has been found and the keyword *on* was expected to be found, instead of which the token *token* was found.

missing TO or FOR for REP statement, found *token*

A replicator statement has been found and the keywords *to* or *for* were expected to be found, instead of which the token *token* was found.

missing] for subscript, found *token*

A subscript operator has been found and a closing square bracket was expected to be found after the subscript value, instead of which the token *token* was found.

missing attributes for attribute list

An attribute list has been found which is empty.

missing constants for constant list

A constant list has been found which is empty.

missing identifier for # directive, found *token*

A # directive has been specified and an identifier was expected to be found after the #, instead of which the token *token* was found.

missing identifier for REP statement, found *token*

A replicator statement has been found and an identifier was expected to be found after the keyword `rep`, instead of which the token *token* was found.

missing identifier for VAL statement, found *token*

A value statement has been found and an identifier was expected to be found after the keyword `val`, instead of which the token *token* was found.

missing identifier for attribute list, found *token*

An attribute list has been found and an identifier was expected to be found after the opening parenthesis starting the list, instead of which the token *token* was found.

missing identifier for attribute, found *token*

An attribute list has been found and an identifier was expected to be found in the attribute list, instead of which the token *token* was found.

missing identifier for name, found *token*

A name expression has been found and an identifier was expected to be found at the start of the expression, instead of which the token *token* was found.

missing identifier for subfield, found *token*

A subfield expression has been found and an identifier was expected to be found after the subfield operator, instead of which the token *token* was found.

missing statements for statement list

A statement list has been found which is empty.

missing string for #INCLUDE statement, found *token*

An `#include` statement has been found and a string was expected to be found after `#include`, instead of which the token *token* was found.

missing string for USE statement, found *token*

A `use` statement has been found and a string was expected to be found after the `use` keyword, instead of which the token *token* was found.

missing type for DEFINE statement, found *token*

A define statement has been found and a type identifier was expected to be found after the keyword `define`, instead of which the token *token* was found.

missing type for attribute, found *token*

A parameter list declaration has been found and a parameter type was expected to be found in the list, instead of which the token *token* was found.

missing } for constant list, found *token*

A constant list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

missing } for statement list, found *token*

A statement list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

modification of '*name*' illegal, already used

The named symbol is a node type that has been used to derive other symbols and an attempt has been made to modify one of its attributes.

object file for '*name*' undefined

Named process has not been associated with an object file.

overflow in REP statement expression

A numerical overflow has occurred during the evaluation of a replicator statement, that is, the replicator identifier has overflowed.

overflow in arithmetic expression

A numerical overflow has occurred during the evaluation of an arithmetic expression.

overflow in decimal integer constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit decimal integer constant.

overflow in dimension size expression

A numerical overflow has occurred during the evaluation of a dimension size expression (which is done to the precision of the hosts integer word length).

overflow in dimension sizes for 'name'

A numerical overflow has occurred during the evaluation of the array size for the named symbol (performed to the precision of the integer word length of the host).

overflow in dimension sizes for constant

A numerical overflow has occurred during the evaluation of the array size for a constant array (performed to the precision of the integer word length of the host).

overflow in hexadecimal integer constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 bit signed hexadecimal integer constant.

overflow in octal integer constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 signed bit octal integer constant.

overflow in real double constant

A numerical overflow has occurred during the conversion of a string of digits representing a 64 bit real constant.

overflow in real float constant

A numerical overflow has occurred during the conversion of a string of digits representing a 32 bit real constant.

overflow in subscript value expression

A numerical overflow has occurred during the evaluation of a subscript value expression (which is done to the precision of the hosts integer word length).

place 'name' on 'name' illegal, channel/edge

An illegal **place** statement has been specified where the first named element is a channel and the second named element is an input (or output) edge.

place 'name' on 'name' illegal, edge/link

An illegal **place** statement has been specified where the first named element is an input (or output) edge and the second named element is a link.

process 'name' and channel 'name' placed on different processors

The named channel, which is a channel of the named process, has been placed on the link of a processor which is different to the processor placed with the process.

process 'name' and processor 'name' execution types mismatch

The named process has an execution type (specified in the object file associated with the process) which is incompatible with the execution types of other processes executing on the named processor.

process 'name' and processor 'name' processor mismatch

The named process has a processor type (specified in the object file associated with the process) which is incompatible with the processor type of the named processor.

process 'name' multiply USED

Named process has been used more than once in a **use** statement.

process 'name' multiply placed

Named process has been used more than once in a **place** statement.

process 'name' unplaced

Named process has not been placed.

process type 'name' multiply USED

Named process type has been used more than once in a **use** statement.

processes 'name' and 'name' placed on different processors

The named processes, which are connected by channels, have been placed onto different processors and there is no link connection is available between the processors for placing the channels.

processor '*name*' unconnected and placed

The named processor has not been connected to the hardware network and has been placed with one or more processes.

reference to undefined symbol '*name*'

Named symbol has been referenced but had not been defined at the point of reference.

root processor '*name*' undefined

When configuring to boot from ROM, the named processor (specified using the **P** option) has not been defined in the configuration source.

subscript out of range for '*name*', *value*

Named symbol has been accessed with the subscript operator and the subscript value used is outside the valid range of the dimension being subscripted. *value* is the dimension number that was subscripted.

subscript out of range for constant, *value*

A constant value has been accessed with the subscript operator and the subscript value used is outside the valid range of the dimension being subscripted. *value* is the dimension number that was subscripted.

symbol '*name*' multiply defined in symbol table

Named symbol has been multiply defined in the configuration source.

unaligned address for attribute '*name*'

The named attribute, which is a sub-attribute of the **location** attribute for a process, has been assigned an address which is not word aligned. *value* is the unaligned address.

uninitialised symbol '*name*' in expression

Named symbol, which is of arithmetic type, has been used in an expression and has not been assigned any value.

unterminated character constant

A character constant has been specified where a closing quote has not been found before the end of the line.

unterminated comment statement

A comment has been started and has not been terminated before the end of the file.

unterminated string constant

A string constant has been specified where a closing double quote has not been found before the end of the line.

unused connector '*name*' in placement

Named connector has not been used in a **connect** statement and has been used in a **place** statement.

value for attribute '*name*' out of range

Named attribute has been assigned a value that is not in the valid range for the attribute.

zero length character constant

A zero length character constant has been specified.

***string* for process '*name*' exceeds maximum memory address**

The named segment of the named process has been specified an address which results in the segment exceeding the maximum memory address of the processor that the process has been placed on. *string* is the memory segment name which can be **code**, **heap**, **stack**, **static** or **vector**.

***string* for process '*name*' overlaps unusable memory**

The named segment of the named process has been specified an address which results in the segment overlapping the unusable memory region of the processor that the process has been placed on. *string* is the memory segment name which can be **code**, **heap**, **stack**, **static** or **vector**.

2.4.4 Serious messages

The following diagnostic messages are generated at severity level *Serious*.

ROM memory size required when booting from ROM

The **RA** or **RO** command line options have been specified and the **RS** option has been omitted.

TCOFF descriptor, illegal dimension size, *value*

TCOFF descriptor, illegal type for *name*, *type*

TCOFF descriptor, missing (, found *char*

TCOFF descriptor, missing), found *char*

TCOFF descriptor, missing :, found *char*

TCOFF descriptor, missing ? or !, found *char*

TCOFF descriptor, missing], found *char*

TCOFF descriptor, missing OCCAM PROC keyword
 TCOFF descriptor, missing OCCAM identifier
 TCOFF descriptor, missing OF for CHAN or PORT parameter
 TCOFF descriptor, overflow in dimension size
 TCOFF descriptor, undefined channel parameter
 TCOFF descriptor, unknown OCCAM parameter type
 TCOFF descriptor, unknown OCCAM process type
 TCOFF format, expected INDEX-ENTRY command (*value*)
 TCOFF format, expected LIB-INDEX-START command (*value*)
 TCOFF format, expected LINKED-UNIT command (*value*)
 TCOFF format, expected START-MODULE command (*value*)
 TCOFF format, invalid ADJUST-POINT adjust size (*value*)
 TCOFF format, invalid ADJUST-POINT value type (*value*)
 TCOFF format, invalid DEFINE-MAIN symbol reference (*value*)
 TCOFF format, invalid DEFINE-MAIN/DESCRIPTOR definitions
 TCOFF format, invalid DEFINE-SYMBOL symbol reference (*value*)
 TCOFF format, invalid DEFINE-SYMBOL value type (*value*)
 TCOFF format, invalid DESCRIPTOR language type (*value*)
 TCOFF format, invalid DESCRIPTOR scalar size (*value*)
 TCOFF format, invalid DESCRIPTOR string size (*value*)
 TCOFF format, invalid DESCRIPTOR symbol reference (*value*)
 TCOFF format, invalid DESCRIPTOR vector size (*value*)
 TCOFF format, invalid INDEX-ENTRY attributes (*value*, *value*)
 TCOFF format, invalid INDEX-ENTRY language type (*value*)
 TCOFF format, invalid INDEX-ENTRY string size (*value*)
 TCOFF format, invalid LOAD-TEXT text size (*value*)
 TCOFF format, invalid ORIGIN SYMBOL format ('*string*')
 TCOFF format, invalid SET-LOAD-POINT symbol reference (*value*)
 TCOFF format, invalid START-MODULE attributes (*value*, *value*)
 TCOFF format, invalid START-MODULE language type (*value*)
 TCOFF format, invalid SYMBOL string size (*value*)
 TCOFF format, invalid code entry offset (*value*)
 TCOFF format, invalid/undefined DEFINE-MAIN definition
 TCOFF format, multiple DEFINE-MAIN commands
 TCOFF format, multiple DESCRIPTOR commands
 TCOFF format, multiple LOAD-TEXT commands
 TCOFF format, multiple ORIGIN SYMBOL commands
 TCOFF format, multiple VIRTUAL SECTION commands
 TCOFF format, unexpected ADJUST-POINT command
 TCOFF format, unexpected command (*value*)

An error has been detected in an object file specified by a `use` statement or a library file containing the system processes.

TCOFF format, expected INDEX-ENTRY command (*value*)

A module in a library file containing the system processes has been requested and does not exist.

advanced and interactive/postmortem debugging are incompatible

The GA option and the G or GP options have been specified together.

advanced debugging and profiling are incompatible

The GA option and the PRE or PRU options have been specified together.

advanced debugging requires software through routing

The GA option and the NV option have been specified together.

booting from ROM and advanced/interactive debugging are incompatible

The G or GA options and the RA or RO options have been specified together.

booting from ROM and profiling are incompatible

The PRE or PRU options and the RA or RO options have been specified together.

execution and utilisation profiling are incompatible

The PRE option and the PRU option have been specified together.

illegal ROM memory size, *value*

Value specified for the RS option is not greater than zero. *value* is the illegal memory size.

illegal format ROM memory size, *string*

An illegal format memory size value has been specified for the RS option. *string* is the illegal format memory size.

illegal record length (*value*)

A record length has been input from a file which exceeds the maximum string length for a file. *value* is the illegal record length found.

illegal string length (*value*)

A string length has been input from a file which exceeds the maximum record length for a file. *value* is the illegal string length found.

interactive and postmortem debugging are incompatible

The G option and the GP option have been specified together.

internal token buffer overflow, *value*

An internal buffer used for storing a source line has overflowed. *value* is the size of the internal buffer in bytes.

multiple ROM memory sizes, *string*

The RS option has been specified more than once. *string* is the latest value for the RS option.

multiple input file names, *string*

The input file name has been specified more than once. *string* is the latest input file name.

multiple output file names, *string*

The O option has been specified more than once. *string* is the latest value for the O option.

multiple processor names, *string*

The P option has been specified more than once. *string* is the latest value for the P option.

processor name required when booting from ROM

The RA or RO options have been used and no P option has been specified.

running from ROM and post mortem debugging are incompatible

The RO option and the GP option have been specified together.

too many errors occurring, *value*

Number of errors exceeds maximum number allowed. *value* is the maximum number of errors allowed.

unable to allocate memory

Amount of memory available to the configurer is insufficient for configuring the configuration source.

unable to close '*string*' (*value*)**unable to close (*value*)****unable to open '*string*' (*value*)****unable to open (*value*)****unable to read (*value*)****unable to seek (*value*)****unable to tell (*value*)****unable to write (*value*)**

These messages are generated as a result of an error occurring in the host file system. *value* is the error failure code.

unexpected command line token, *string*

An argument has been specified on the command line to the configurer that is not recognized as a valid option string.

unexpected end of input

The end of the file has been found unexpectedly in an object file.

2.4.5 Fatal errors

Any fatal errors which occur should be reported to your local INMOS distributor or field applications engineer.

The following errors are generated at severity *Fatal*:

**did not find all processors in BuildDataStructs()
did not find all processors in FillInKernelTable()
did not find all processors in PlaceDebugKernels()**

An internal error has occurred in the configurer. The configurer has found an internal inconsistency while virtual routing.

problem in allocation routines

An internal error has occurred in the configurer. The configurer has incorrectly attempted to allocate memory from the heap.

problem in deallocation routines

An internal error has occurred in the configurer. The configurer has incorrectly attempted to return memory to the heap.

**unable to support advanced debugger, no host edge
unable to support interactive debugger, no host edge**

An internal error has occurred in the configurer.

3 `icollect` — code collector

This chapter describes the code collector tool `icollect` which generates an executable file for a single or multitransputer program from a configuration data file, or for a single transputer program directly from a linked unit. The tool is also used to create files for input to the EPROM programmer tool `ieprom`, and to create files that can be dynamically loaded by a user program.

3.1 Introduction

`icollect` generates bootable files for transputer programs, and other executable files in special formats.

Bootable files are transputer executable files that can be directly loaded onto the transputer hardware down a transputer link. The bootable file contains all the information for loading and running the program on a specific network of processors, including data that controls the distribution of code on the network, and self-booting code for each processor. Bootable programs are therefore self-distributing and self-starting when they are sent down a transputer link.

Recommended program development for single and multitransputer programs is to create a configuration data file (i.e. binary file) and to use this as input to the collector. The configuration data file describes the placement of processes and channels on the processor network in a special format which can be read by the collector. They are created from configuration descriptions by the configurator.

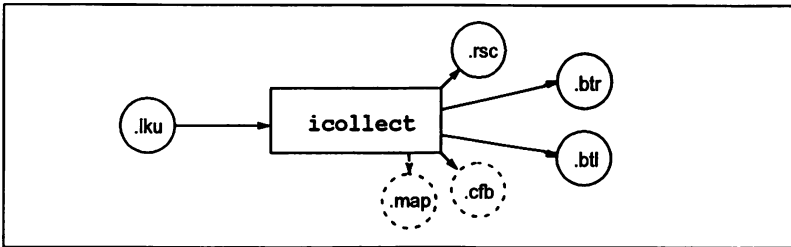
Single transputer programs can by-pass the configuration stage and use a single linked unit as input. The collector then adds bootstrap and system code for a single processor. Unconfigured programs can only run on a single transputer.

`icollect` can be directed to generate output files in a special format for processing by the `ieprom` tool, and executable code with no bootstrap or system process information, intended for dynamic loading by a supervisory program.

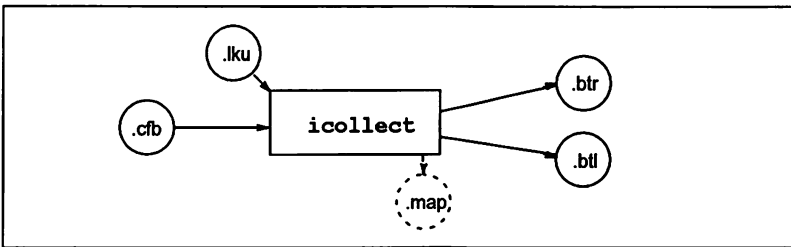
The command line default is to assume input from a configuration binary file. Special format outputs are selected by specifying command line options.

The main inputs and outputs of the collector tool for bootable programs are shown below.

Unconfigured program (using 'T' option):



Configured processor program:



3.2 Running the code collector

The code collector is invoked using the following command line:

► `icollect filename { options }`

where: *filename* is a configuration data file created by a configurer or a single linked unit created by `ilink`. Only one filename may be given on the command line.

options is a list of the options given in Table 3.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
B <i>filename</i>	<p>Uses a user-defined bootstrap loader program in place of the standard bootstrap. The program is specified by <i>filename</i> and must conform to the rules described in appendix F.</p> <p>This option can only be used with the 'T' option (unconfigured mode) and cannot be used with the 'RA' and 'RO' options.</p>
BM	<p>Instructs the tool to use a different bootstrapping scheme, which uses the bottom of memory, see section 3.8.</p> <p>This option is only valid for configured programs i.e. when the 'T' option is <i>not</i> used.</p>
C <i>filename</i>	<p>Specifies a name for the debug data file. A filename must be supplied and is used as given.</p> <p>This option can only be used with the 'T' option (unconfigured mode) and cannot be used with the 'D' or 'K' options.</p>
CM	Instructs the collector to add a bootstrap which will clear memory during the booting and loading of the transputer network. Intended for use with parity-checked memory (see section 3.4).
D	Disables the generation of the debug data file for single transputer programs. This option can only be used with the 'T' option (unconfigured mode).
E	<p>Changes the setting of the transputer Halt On Error flag. HALT mode programs are converted so that they not stop when the error flag is set, and non HALT mode programs to stop when the error flag is set.</p> <p>This option can only be used with the 'T' option (unconfigured mode).</p>
I	Displays progress information as the collector runs.
K	<p>Creates a single transputer file with no bootstrap code. If no file is specified the output file is named after the input filename and given the <i>.rsc</i> extension.</p> <p>This option can only be used with the 'T' option (unconfigured mode).</p>
M <i>memorysize</i>	<p>Specifies the memory size available (in bytes) on the root processor for single transputer programs. <i>memorysize</i> is specified in bytes and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilobytes or Megabytes respectively), or it may be specified in hexadecimal using the '#' or '\$' prefixes.</p> <p>This option can only be used with the 'T' option (unconfigured mode) and results in a smaller amount of code being produced (see section 3.3).</p>

Option	Description
O <i>filename</i>	Specifies the output file. A filename must be supplied and is used as given. (See section 3.2.4).
P <i>filename</i>	Specifies a name for the memory map file. A filename must be supplied and is used as given. The file extension <i>.map</i> should be used when the file is to be used as input to <i>imap</i> , see chapter 12.
RA	Creates a file for processing by <i>ieprom</i> into a boot from ROM file to run in RAM. If no output file is specified the filename is taken from the input file and given the <i>.btr</i> extension. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.
RO	Creates a file for processing by <i>ieprom</i> into a boot from ROM file to run in ROM. If no output file is specified the filename is taken from the input file and given the <i>.btr</i> extension. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.
RS <i>romsize</i>	Specifies the size of ROM on the root processor in bytes. Only valid when used with the 'RA' or 'RO' options. <i>romsize</i> is specified in bytes and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilobytes or Megabytes respectively), or it may be specified in hexadecimal using the '#' or '\$' prefixes. This option is only necessary when using the 'T' option (unconfigured mode) to create a ROM code file.
S <i>stacksize</i>	Specifies the extra runtime stack size in words for single transputer programs. <i>stacksize</i> is specified in words and may be given in decimal format (optionally followed by 'K' or 'M' to indicate Kilowords or Megawords respectively), or it may be specified in hexadecimal using the '#' or '\$' prefixes. This option can only be used with the 'T' option.
T	Creates a bootable file for a single transputer. The input file specified on the command line must be a linked unit. This option can not be used for programs linked with the <i>reduced</i> runtime library.
Y	Disables interactive debugging with <i>idebug</i> and reduces the amount of memory used. (See section 3.10). This option can only be used with the 'T' option (unconfigured mode).

Table 3.1 *icollect* command line options

3.2.1 Examples of use

Example A (unconfigured program mode):

UNIX based toolsets:

```
icc hello
ilink hello.tco -f cnonconf.lnk
icollect hello.lku -t
iserver -sb hello.btl -se
```

MS-DOS/VMS based toolsets:

```
icc hello
ilink hello.tco /f cnonconf.lnk
icollect hello.lku /t
iserver /sb hello.btl /se
```

Example B (configured program mode):

UNIX based toolsets:

```
icc hello
ilink hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS/VMS based toolsets:

```
icc hello
ilink hello.tco /f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

Note: single transputer programs linked with the *reduced* runtime libraries cannot be linked and collected with the 'T' option, they must be configured.

3.2.2 Default command line

Commonly used command line parameters can be defined for the tool using the `ICOLLECTARG` environment variable. Parameters specified in this way are automatically added to the command line when the tool is run.

Parameters in `ICOLLECTARG` must be specified using the syntax required by the command line.

3.2.3 Input files

The input file to `icollect` is either a configuration data file generated by a configurator, or a linked unit generated by `ilink`. By default the collector assumes a configuration data file; for single transputer programs the input file may be a linked unit, in this case the 'T' option must be given.

Input files of an incorrect format generate an error message and no output is produced.

3.2.4 Output files

The output produced by the tool depends the type of file input to the collector and the collector options used. Provided the user does not specify an output file using the 'o' option, files will be produced with the extensions indicated below:

Single processor non-configured case (T option)

The default output file is a binary file that can be loaded directly onto the transputer hardware down a transputer link. This type of file is known as a *boot from link* program. Boot from ROM programs and dynamically loadable code must be specified as output using the appropriate command line options.

If no filename is specified the output file is named after the input file and given a .bt1 extension. If an output filename is specified the file is given the specified name.

Files created using the 'RA', 'RO', and 'K' options, and where no output filename is specified, are given special extensions to indicate the file type. File extensions used for each of the file types are listed below.

Option specified	File type	Extension given
RA	RAM executable file	.btr
RO	ROM executable file	.btr
K	Dynamically loadable file	.rsc

Configured programs

When the program has been configured, the collector will output a file with the .bt1 extension if the program was configured to boot from link; the default case. If the program was configured to boot from ROM then the collector will generate a .btr file.

Memory map files

A memory map file may be generated, in addition to the normal output, by specifying the 'p' option. The format of these files is described in section 3.9.

Debug data file

For unconfigured transputer programs only, the collector automatically generates a configuration binary file for use by the debugger. By default the filename stem is taken from the output file and the extension '.cxb' is added. If the 'C' option is specified then the filename given is used, as supplied. Generation of the debug data file can be disabled with the 'D' option.

3.3 Memory allocation for unconfigured programs

The memory allocation outlined in this section applies only to single processor programs collected with the 'T' option and without the 'K' option. For configured programs the layout of code and data in memory is determined by the configurator. For

programs generated with the 'T' option the layout is determined by the collector. The details of memory use depend on the language used and the options to `icollect`, this is described below.

Memory which is not reserved by the system for program code and data (known as *free memory*) can be made available to a user application. For C programs this is used for the heap and, optionally, the stack. In the case of a single OCCAM program the free memory passed as an array.

To calculate the actual memory available, the loader program in the bootable file first reads the total memory size from the host environment variable `IBOARDSIZE`. This communication with the host is performed after the program has been loaded onto the transputer board but before the program is started. The size of the free memory is given by `IBOARDSIZE` minus the combined program code and data space required.

The process code which reads `IBOARDSIZE` requires approximately 3.5 Kbytes of memory. This process is executed and terminated before the user program runs, and the segment of free memory that the process uses is then returned to the user program. Therefore when the user program executes it will not know whether the process was present or not.

When the 'M' option is used to specify the memory size, `IBOARDSIZE` is not read and therefore the total amount of memory required when loading the program will be approximately 3.5 Kbytes less.

A memory map file may be obtained by specifying the 'P' command line option. The content of memory map files is described in section 3.9.

3.3.1 C and FORTRAN programs

For C programs the bootstrap loader must allocate memory for static data, stack and heap areas. FORTRAN programs have similar requirements and are handled in the same way.

When the collector 'S' option is specified the program's stack is placed at the bottom of memory. When the 's' option is not specified a stack area is allocated by the runtime system, typically at the top of free memory.

Areas for static data and heap are always allocated by the language's runtime system at the bottom of free memory. The heap area grows upwards, towards the top of memory, and the stack grows downwards.

Figure 3.1 shows the memory map layouts for programs with and without the stack requirement specified by the user.

The value of `LoadStart` is described in section 3.9.

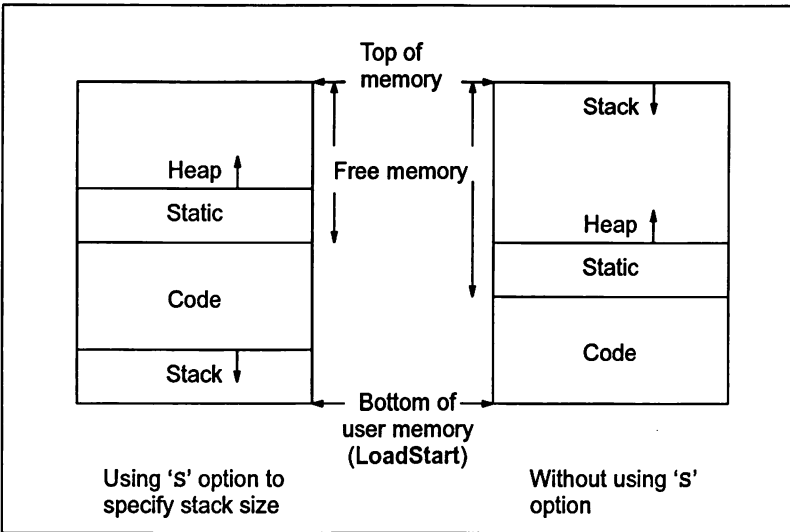


Figure 3.1 Memory maps for C and FORTRAN

3.3.2 occam programs

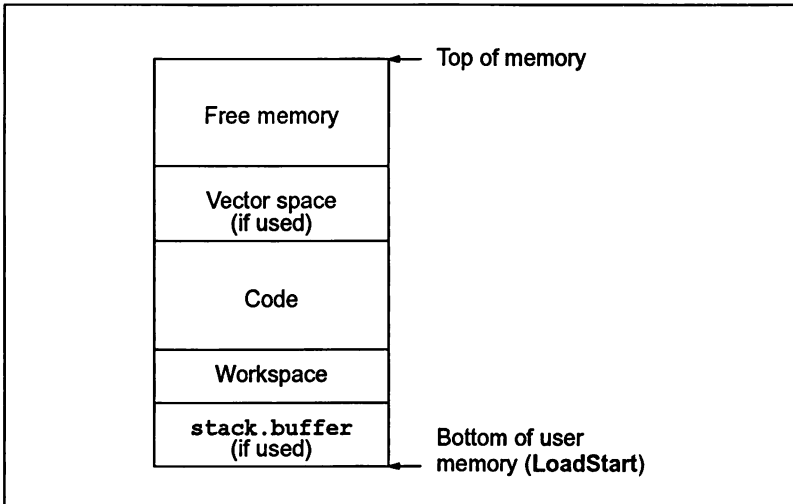


Figure 3.2 Memory map for occam program

An occam program requires space to be allocated for code, workspace and, possibly, vector space. Programs can also be passed one or two arrays as parameters;

one (always available) provides access to the free memory. The other is optional but, if used, it is placed at the bottom of the memory map to provide access to the transputer's fast internal RAM. This array is known as the `stack.buffer`. The default bootstrap loader attempts to optimize placement of the program's, and its own, code and workspace. If present, the `stack.buffer` array is placed at the bottom of memory (at `LoadStart`). This is followed in order by the workspace, code, vector space (if used) and free memory.

Figure 3.2 shows the memory map of the loaded Occam code as created by the default bootstrap loader.

3.3.3 Memory initialization errors

While the loader is executing the memory initialization process, described above, warning messages may be obtained which have the following format:

Warning-SystemA- *message*

where: *message* can be one of the following:

IBOARDSIZE, unable to read

IBOARDSIZE environment variable is not defined correctly.

number, illegal format number

The value specified for IBOARDSIZE is in the wrong format.

illegal 16 bit memory size, set to zero

The value of IBOARDSIZE is greater than 64K when a 16 bit processor is being used. The memory size has therefore been set to zero.

negative memory size, set to zero

A negative value was specified for IBOARDSIZE, which has been set to zero.

unable to reset free memory

The loader cannot return the memory it has used to the user.

All the above errors are generated by the system process at runtime.

3.3.4 Small values of IBOARDSIZE

When the 'T' option is used, very small values of IBOARDSIZE (including zero) are detected at runtime and prevent the program from being run. (This can also happen when IBOARDSIZE has been incorrectly specified, because icollect assumes a default value of zero.) Small values of IBOARDSIZE cause the collector to generate a warning message but do not prevent the generation of a bootable file.

IBOARDSIZE must be \geq to the total memory requirements of the user program being executed.

3.4 Parity-checked memory

If any processors in the network use parity-checking on external memory (typically using the T426) then it is essential that that memory is initialized (written to) before it is read. Reading from an uninitialized location is likely to cause a parity error. Internal memory is not parity-checked, so a bootstrap program can always get started, but the initialization must be done before any program using external memory is run. Therefore the initialization of memory must be done during the booting and loading of the processors.

There is an option to the collector, '**CM**', which instructs the collector to use a bootstrap that clears memory on each transputer before the application code is executed. This must always be used when collecting programs for a network that contains one or more T426s. When selected, the '**CM**' option applies to all processors in the network, not just to T426s.

In order to clear the memory on a processor, it is necessary for the bootstrapping sequence to know the size of the memory. There are four cases to consider:

1 A configured program.

Here the memory size is known at configuration time, and is specified by the user in the configuration source file. The bootstrapping sequence produced by the collector will clear the amount of memory specified (in the configuration source file) before booting the application.

2 A collected program with fixed memory size.

The collector may be used, with the '**T**' option, to produce a bootable file from a single linked unit. The amount of memory on the processor may be specified with the '**M**' option. In this case the bootstrapping sequence will clear the amount of memory specified (with the '**M**' option) before booting the application.

3 A collected program with variable memory size, booted from link.

If the collector is run with the '**T**' option, but without the '**M**' option, the memory size is known only at runtime. The memory size is found out at runtime using the environment variable **IBOARDSIZE**. In this case the bootstrapping sequence will clear memory up to the minimum required to boot the program. After booting, the value of **IBOARDSIZE** will be read and the remaining memory will be cleared.

4 A collected program with variable memory size, booted from ROM.

If the collector is run with the '**T**' option, but without the '**M**' option, and the program is booted from ROM, then the memory size is not known at all. In

this case the bootstrapping sequence will clear enough memory for the minimal requirements of the application. It is then the user program's responsibility to clear any additional memory required.

Initialization of memory is carried out regardless of the processor type; memory is initialized even if the processor is not a T426. So if the 'CM' option is selected every processor's memory in the network will be initialized (including 16-bit transputers). In the case of the T426, the bootstrap code also clears the parity registers by reading them before the program starts.

3.5 Non-bootable files created with the X option

Files created with the 'X' option are non-bootable files which can be dynamically loaded or manipulated by a program at runtime. Non-bootable files cannot be loaded and run on transputer hardware in the normal way.

3.5.1 File format

Non-bootable files consist of program code preceded by a specific sequence of data words which provide runtime information. The sequence of data words and code blocks is summarized in table 3.2. Descriptions of the more important data items are given in table 3.3.

Data	Number of bytes occupied	Unit
Interface descriptor size	Four	bytes
Interface descriptor	Set by above	–
Compiler id size	Four	bytes
Compiler id	Set by above	–
Target processor type	Four	–
Version number	Four	–
Program scalar workspace requirement	Four	words
Program vector workspace requirement	Four	words
Static size	Four	words
Program entry point offset	Four	bytes
Program code size	Four	bytes
Program code block	Set by above	–

Table 3.2 Sequence of code segments in non-bootable files

Target	A value indicating the processor type or transputer class for which the program was compiled. Set by compiler options or by default. Possible values and their meaning are: <table> <tr> <th>Value</th><th>Applies to:</th></tr> <tr> <td>2</td><td>T212, T222, T225, M212</td></tr> <tr> <td>4</td><td>T414</td></tr> <tr> <td>8</td><td>T800, T801, T805</td></tr> <tr> <td>9</td><td>T425, T400, T426</td></tr> <tr> <td>10</td><td>TA</td></tr> <tr> <td>11</td><td>TB</td></tr> </table>	Value	Applies to:	2	T212, T222, T225, M212	4	T414	8	T800, T801, T805	9	T425, T400, T426	10	TA	11	TB
Value	Applies to:														
2	T212, T222, T225, M212														
4	T414														
8	T800, T801, T805														
9	T425, T400, T426														
10	TA														
11	TB														
Version	The format version number of the file. This can be either 10 or 11 in TCOFF files. For C and FORTRAN programs this value is 11, which indicates that the 'Static size' parameter (below) is present. For occam programs the value is 10, indicating no static data; the parameter list will also not be present.														
Scalar workspace	Specifies the size of the workspace required for the linked program's runtime stack.														
Vector workspace	Specifies the size of the workspace required for the linked program's vector (array) data.														
Static size	Specifies the size of the static area (only present if the file format version number is 11).														
Entry point offset	Indicates the offset in bytes of the program entry point from the base of the code block.														
Code size	Indicates the size of the program code in bytes.														
Code	The program code.														

Table 3.3 Details of code segments in non-bootable files

3.6 Boot-from-ROM output files

Boot-from-ROM output files are either generated by using the collector options 'RA' or 'RO' for unconfigured programs or by configuring a program to boot from ROM, prior to collecting. (The configurator also has 'RA' and 'RO' command line options).

The boot-from-ROM files contain code that can be loaded into EPROM using the `ieprom` tool. The code may be run on the root transputer of a network; processors on the network connected to the root transputer are booted from the root transputer links.

'RA' generates code which is executed from RAM. The code is copied from ROM into RAM at runtime. 'RO' generates code which is directly executed from ROM.

RAM executable code can be used for applications which are to be executed from fast RAM, and for code which may be user-modified. ROM executable code requires no external RAM and can be used to create a truly embedded system.

3.7 Alternative bootstrap loaders for unconfigured programs

If not otherwise specified, `icollect` uses the standard INMOS primary bootstrap loading sequence. The correct code for the application program is chosen automatically from a library of bootstraps compiled for different transputer types and error modes.

The collector can be directed to use other bootstrap loader programs defining different loading sequences by specifying the 'B' option. This option directs the collector to append a user-defined loader program in place of the standard bootstrap code.

User-defined bootstraps must comply with the format used by the standard INMOS loader. The source of the standard INMOS Network Loader is supplied with the toolset. The source is fully commented and can be used as a template to design and code your own loading sequence.

3.8 Alternative bootstrap schemes

When building for a configured network, the collector uses a bootstrapping scheme which makes use of the top two hundred bytes of memory. This memory is required to load the last few bytes of application code prior to its execution. The memory region becomes available to the user once their application is running.

This scheme does not remove memory from the user's environment on a permanent basis and it facilitates the absolute placement of code and data by the user. See the *Toolset User Guide* for details.

The user can tell the collector to use a different booting scheme by using the option 'BM'. In this case the booting scheme permanently removes a section of memory from the user's environment and moves the value of `LoadStart` accordingly. This section of memory is never made available to the user. This booting scheme does not support the absolute placement of code and data by the user.

The booting scheme invoked by the 'BM' option, is used by default for unconfigured programs i.e. those collected using the 'T' option.

3.9 The memory map file

A memory map file may be obtained by specifying the 'P' command line option, followed by a filename. Such files contain the memory layout for each processor in the network.

The file layout takes the form of a list of code and data to be placed on respective processors. The right hand side of the file gives the start and end address followed by the size of each block.

The memory map file contains the following information:

- **icollect** version data
- For each processor the following details are given:
 - Processor type
 - Error mode (HALT or STOP)
 - **LoadStart** (lowest user memory address)
 - For each process on this processor the following is listed:
 - Code, name of file, offset from start (decimal), start address and end address (hex), size (decimal), entry address (if any, in Hex)
 - Workspace, start and end address (hex), size (decimal)
 - Any other data requirements
- Boot path for the network - only present if program is configured
- Connectivity of the network - only present if program is configured

The absolute addresses are calculated using **LoadStart**, which is the base of user memory. This varies for different processor types i.e. the value of **LoadStart** for a T4 processor is different to that for a T8.

If the '**BM**' option is used the memory from **MemStart** to **LoadStart** is used by the low level bootstraps and their workspace.

When the '**BM**' option is not used the value of **LoadStart** is determined by the configuration, see the reference chapter for the configurator, for further details.

The addresses allocated to various data items reflect the command line options specified to the collector. Details of the memory map files for the following types of files are given below:

- Unconfigured (single processor), boot from link programs targetted at a specific processor type.
- Unconfigured (single processor), boot from link programs targetted at a processor class.
- Configured, boot from link programs.
- Boot from ROM (single and configured)

3.9.1 Unconfigured (single processor), boot from link

Program targetted at transputer type

The first memory map described in this section is for a program which is to be booted for a specific processor type.

The example shown in Figure 3.3 was produced by the following command line:

```
icollect -t hello.lku -s 400 -p hello.map (UNIX)
```

```
icollect /t hello.lku /p /s 400 hello.map (MS-DOS/VMS)
```

where: `hello.lku` was produced by compiling and linking the example program `hello.c` for a T425 in the default halt-on-error mode. The compiled object file was linked with the C linker indirect file `cnonconf.lnk` because the example is for an unconfigured program.

`hello.map` lists code and data segments to be placed on each processor (one in this case). For each process the workspace and vector space requirements are given together with the entry point of the process. Notice that the first three processes listed are non-user processes; this will always be the case for this type of program.

```
icollect : INMOS toolset collector
Sun Version 3.0.14

Memory map for processor 0 T425
Load Start is 80000168, HALT ON ERROR, Minimum memory size is 21056
LOW priority INITSYSTEM process 'Init.system'
  Code from 'sysproc.lib', file offset 9438
    Entry address      #800001F8 #80000418      544
    Invocation stack   #800001F9 #800001D8      24
    Workspace          #800001D8 #800001D8      112

LOW priority SYSTEM process 'System.process.a'
  Code from 'sysproc.lib', file offset 27180
    Entry address      #80004670 #80005040      2512
    Invocation stack   #80004671 #80004668      20
    Workspace          #80004654 #80004654      536
    Vectorspace        #8000443C #80005240      512

HIGH priority SYSTEM process 'System.process.b'
  Code from 'sysproc.lib', file offset 45498
    Entry address      #8000044C #800004A8      92
    Invocation stack   #8000044C #80000444      20
    Workspace          #80000430 #80000430      24

LOW priority USER process
  Code from 'hello.lku', file offset 2
    Entry address      #80000888 #8000427C      14836
    Invocation stack   #800008B3 #80000880      20
    Workspace          #8000086C #8000086C      196
    Extra stack        #800007A8 #800007A8      1600
    Static              #80000168 #8000466A      558
    Parameter data     #8000443C #8000443C      448
```

Figure 3.3 Memory map file for a single T425 processor program

Program targetted at transputer class

The second memory map described in this section is for a program which is to be booted for processor classes TA or TB.

The example shown in Figure 3.4 was produced by the following command line:

```
icollect -t hello.lku -p hello.map      (UNIX)
```

```
icollect /t hello.lku /p hello.map     (MS-DOS/VMS)
```

where: **hello.lku** was produced by compiling and linking the example program **hello.c** for class TA in the default halt-on-error mode. The compiled object file was linked with the C linker indirect file **cnonconf.lnk** because the example is for an unconfigured program.

```
icollect : INMOS toolset collector
Sun Version 3.0.14

Memory map for processor 0 TA
Load Start is UNKNOWN, HALT ON ERROR, Minimum memory size is 20180
  LOW priority INITSYSTEM process 'Init.system'
    Code from 'sysproc.lib', file offset 10420
      Entry address      #3D48      #3F68      544
      Invocation stack   #3D49
      Workspace          #3D28      #3D40      24
      Workspace          #3CB8      #3D28      112

  LOW priority SYSTEM process 'System.process.a'
    Code from 'sysproc.lib', file offset 30561
      Entry address      #419C      #4B6C      2512
      Invocation stack   #419D
      Workspace          #4180      #4194      20
      Workspace          #3F68      #4180      536
      Vectorspace        #4B6C      #4D6C      512

  HIGH priority SYSTEM process 'System.process.b'
    Code from 'sysproc.lib', file offset 45888
      Entry address      #34         #90         92
      Invocation stack   #34
      Workspace          #18         #2C         20
      Workspace          #0          #18         24

  LOW priority USER process
    Code from 'hello.lku', file offset 2
      Entry address      #E0         #3AF8      14872
      Invocation stack   #10B
      Workspace          #C4         #D8         20
      Workspace          #0          #C4         196
      Static             #3F68      #4196      558

Parameter data          #3AF8      #3CB8      448
```

Figure 3.4 Memory map file for a single TA processor program

The memory layout is the same as for the previous example except that no space is allocated for the extra stack (because extra stack was not requested on the command line). **LoadStart**, from which the start and end addresses are calculated, can

only be calculated at runtime. This is because the value of **MemStart** cannot be determined at collect time. The numbers given, in place of absolute addresses are offsets from **LoadStart**.

3.9.2 Configured program boot from link

```

icollect : INMOS toolset collector
Sun Version 3.0.14

Memory map for 'Single' processor 0 T425
Load Start is 800000A0, HALT ON ERROR, Minimum memory size is 73236

HIGH priority INITSYSTEM process 'Init.system.simple'
Code from 'sysproc.lib', file offset 13366
    Entry address      #800000E4 #80000158      116
    Invocation stack   #800000C0 #800000E4      36
    Workspace          #800000A0 #800000C0      32

HIGH priority OVERLAYED SYSTEM process 'System.process.b'
Code from 'sysproc.lib', file offset 44718
    Entry address      #80000184 #800001E0      92
    Invocation stack   #80000170 #80000184      20
    Workspace          #80000158 #80000170      24

LOW priority USER process 'Simple'
Code from 'hello.lku', file offset 2
    Entry address      #80001178 #80004B6C     14836
    Invocation stack   #800011A3 #80001164      20
    Workspace          #800000A0 #80001164     4292
    Static             #80004B6C #80005424     2232
    Heap               #80005424 #80011C24     51200
    Parameter data     #80011C24 #80011D50      300

Boot path for network

Boot processor 0 down link 0 from HOST

Connectivity for network

Connect HOST to processor 0 link 0

```

Figure 3.5 Memory map file for a configured T425 processor program

The example shown in Figure 3.5 was produced by the following command line:

```

icollect hello.cfb -p hello.map      (UNIX)
icollect hello.cfb /p hello.map      (MS-DOS/VMS)

```

where: **hello.cfb** is the configuration binary file produced by the configurator for the single processor 'Hello World' example program introduced in chapter 4 of the *ANSI C Toolset User Guide*.

The Memory map for the configured program is similar to that produced for unconfigured transputer programs except that it has two additional configuration sections at the end of the file. The *Boot path* for the network lists processors in the order in which they are to be booted. The *Connectivity for network* lists the link connections between the processors.

3.9.3 Boot from ROM programs

There are four cases for this type of program:

- Unconfigured (single processor), boot from ROM, run in RAM
- Unconfigured (single processor), boot from ROM, run in ROM
- Configured program, boot from ROM, run in RAM
- Configured program, boot from ROM, run in ROM

The memory maps for each of these are summarized below.

Unconfigured (single processor), boot from ROM, run in RAM

The memory map for this case will have the same layout as the single processor boot from link programs.

Unconfigured (single processor), boot from ROM, run in ROM

It is not known at collect time where in memory the ROM is to be placed. Therefore, the start and end addresses of the code segments are given as offsets from the start of ROM, and are annotated as such. Items such as workspace will have absolute addresses allocated, if the program is targetted at a specific processor type.

Note: for C programs the runtime startup system would require modification first, in order to provide the system with details of heap and stack etc.

Configured program, boot from ROM, run in RAM

The layout of the memory map for this case will be the same as that for the boot from link configured program. This is because everything (code and data) is copied into RAM.

Configured program, boot from ROM, run in ROM

For this case the root processor will be shown in the same format as the single processor case, run in ROM; some memory locations being expressed as offsets from the beginning of ROM.

The other processors in the network will appear as in the boot from link case.

The example shown in Figure 3.6 was produced by the following command line:

```
icollect hello.cfb -p hello.map          (UNIX)
icollect hello.cfb /p hello.map         (MS-DOS/VMS)
```

where: **hello.cfb** is the configuration binary file produced by the configurer, for the single processor 'Hello World' example program introduced in chapter 4 of the *ANSI C Toolset User Guide*. The configurer 'RO', 'RS' and 'P' options were used to create a boot from ROM input file for the collector.

```
icollect : INMOS toolset collector
Sun Version 3.0.14

Memory map for 'Single' processor 0 (Booting and running in ROM) T425
Load Start is 800000A0, HALT ON ERROR, Minimum memory size is 58204

HIGH priority INITSYSTEM process 'Rom.init.system.simple'
Code from 'sysproc.lib', file offset 16750
ROM OFFSET          #3AD3          #3B6F          156
ROM entry offset    #3AD6
Invocation stack    #800000C0 #800000E4          36
Workspace           #800000A0 #800000C0          32

HIGH priority OVERLAYED SYSTEM process 'System.process.b'
Code from 'sysproc.lib', file offset 44718
ROM OFFSET          #3B6F          #3BCB          92
ROM entry offset    #3B6F
Invocation stack    #800000FC #80000110          20
Workspace           #800000E4 #800000FC          24

LOW priority USER process 'Simple'
Code from 'hello.lku', file offset 2
ROM OFFSET          #DF          #3AD3          14836
ROM entry offset    #10A
Invocation stack    #80001164 #80001178          20
Workspace           #800000A0 #80001164          4292
Static              #80001178 #80001A30          2232
Heap                #80001A30 #8000E230          51200

Parameter data      #8000E230 #8000E35C          300

Boot path for network

Connectivity for network
```

Figure 3.6 Memory map for program configured to boot from and run in ROM

3.10 Disabling interactive debugging – 'Y' option

The 'Y' collector option has two effects on the program being built:

- It disables interactive (breakpoint) debugging of the program
- It reduces the amount of memory used.

For programs compiled and linked for a specific transputer type, this option will cause `icollect` to produce a program that uses less memory. However, programs compiled and linked for transputer classes 'TA' or 'TB' will not build when this option is used. This option is only valid for programs collected with the `T` option.

3.11 Error messages

This section lists error messages generated by `icollect`. The messages are listed in alphabetical order under the appropriate severity classification. In all cases the introductory string (severity, and filename if appropriate) is omitted.

`icollect` generates errors of severities *Warning* and *Serious*. Serious errors cause the tool to terminate without producing any output.

3.11.1 Warnings

The following messages are prefixed with 'Warning-'. They are only generated when the `T` option is used (single processor mode).

Extra disable option on command line ignored

The program has been configured with interactive debugging disabled and the `'Y'` option specified to the collector is therefore superfluous.

Flip error mode ignored with user bootstrap

The `'E'` option is ignored when a user-defined bootstrap is specified since the collector will only accept a single linked unit as a bootstrap.

Program configured with interactive debugging enabled, option ignored

The program has been configured with interactive debugging enabled and the `'Y'` option has been specified to the collector. The `'Y'` option is ignored and the boot file is built.

Strange board size for sixteen bit processor: Setting to zero

The memory size specified exceeds the addressing capacity of a 16 bit processor (64 Kbytes). The collector uses a memory size of zero for the rest of the build.

3.11.2 Serious errors

The following errors are prefixed with 'Serious-'.

Address space for target processor exhausted

The address space required by the program is greater than 64Kbytes, the maximum addressable space on a 16-bit processor.

Bootstrap file already specified

More than one bootstrap file was specified. Only one file is allowed.

Bootstrap filename too long

The maximum length allowed for the bootstrap filename is 255 characters.

Bootstrap is greater than 255 byte in library file

The library bootstrap is too large. This should only occur if the library file is invalid or corrupt.

Cannot have both rom types

'RA' and 'RO' options are mutually exclusive and cannot both be specified on the same command line.

Cannot have configured and memory size

The memory size option is incompatible with building a bootable program for a configuration binary file.

Cannot have configured and non bootable file

The collector cannot generate both a network loadable file and a non-bootable file simultaneously for the same program.

Cannot have rom and non bootable file

The collector cannot generate both a ROM-loadable file and a non-bootable file simultaneously for the same program.

Cannot open file *filename*

Host file system error. The file specified cannot be opened.

Cannot patch parameter data for processor class

The 'x' option has been specified with a linked unit for a processor class. The collector cannot initialize some of the data without a linked unit for a specific processor type.

Cannot use absolute placement and bottom of memory loader

The user has specified BM to the collector but is using absolute code and data placement at configuration. This combination is not legal.

Command line parsing error at *string*

Unrecognized command line option.

Debug file already specified

More than one debug was file specified. Specify one only.

Dynamic memory allocation failure

Memory allocation error. The collector cannot allocate the required amount of memory for its internal data structures.

Error in writing to debug file

Host file system error. The debug file could not be written. This message will only appear if the collector is invoked with the 'T' option (unconfigured mode).

Expected end tag found not present in .cfb file

A specific end tag is missing in the configuration binary file. Either the file is corrupted or the versions of `icollect` and `configurer` used are incompatible.

Illegal tag found in .cfb file

Incorrect format configuration binary file, recognized as an illegal tag. Either the file is corrupted or the versions of `icollect` and `configurer` used are incompatible.

Illegal language type found in input file

Source language used to create the file is not supported by the collector. Less likely, but possible, is that the file was created using an incompatible (possibly earlier) version of a tool.

Illegal process type

Unrecognized process type. Either the file has been corrupted or the versions of `icollect` and `configurer` used are incompatible.

Illegal processor type

Unrecognized processor type. Either the file has been corrupted or `icollect` and the `configurer` are incompatible.

Illegal tag found in input file : *filename*

Incorrect format input file. The most likely reason for this error is that an incorrect file has been specified. Other less likely but possible explanations are that the file was created using an earlier or incompatible version of one of the tools, or that the file has become corrupted.

Input file already specified

More than one input file specified on the command line.

Input file has not been linked *filename*

The collector accepts only linked files, either directly when using single processor operation, or indirectly via entries in the configuration binary

file. This message can be generated if the file was created using a previous version of a tool, or if the file is corrupt.

Input file is of incorrect type: *filename*

If the 'T' option is specified (single processor program) the input file must be a single linked unit (.lku type). If the 'T' option has not been specified the input file must be a configuration binary file (.cfb type).

Input filename too long

The maximum length allowed for the input filename is 256 characters.

Linked unit file in cfb and linked unit in input file found do not match: *filename*

The linked file specified in the configuration binary and the one found the collector do not match.

Linked unit module not found in: *filename*

The required library module is missing or has been corrupted. This message is generated when an incorrect version of the library is installed.

Memory requirement for build is greater than specified, an extra <n> bytes required at least

The amount of memory specified on a processor is not enough for the program to execute. An extra <n> bytes are required at least.

Memory size already specified

Memory size must be specified once only.

Memory size string invalid

Memory size must be given in decimal or hex. Hex numbers must be introduced by '#' or '\$'.

Memory size string too long

Specified memory size is too large.

More than one parameter statements

The collector expects only one *parameter* statement per processor. Either the file has been corrupted or the versions of **icollect** and **configurer** used are incompatible.

No debug and debug output file specified in command line

Options 'D' (disable debug) and 'C' (debug filename) cannot be used together.

No input file specified

One, and only one, input file must be specified on the command line.

No parameter descriptor present in input file: *filename*

The formal parameter descriptor in the input file is not present. This usually means that the process has not been linked with a main entry routine. This message will only appear if the collector is invoked with the 'T' option (unconfigured mode).

Output file already specified

More than one output file was specified. Specify only one.

Output filename too long

The maximum length allowed for the output filename is 256 characters.

Parameter descriptor error in input file : *filename*

The formal parameter descriptor in the input file is not of the correct form, indicating that the process interface is not one recognized by the collector. This message will only appear if the collector is invoked with the 'T' option (unconfigured mode).

Print map file already specified

More than one print map file was specified. Specify one only.

Program configured for boot from ROM command line is boot from link

The specified configuration binary file was created for either ROM or RAM, and neither has been specified to `icollect`.

Program configured for running in RA mode command line is RO mode

Wrong mode specified, or incorrect option given to the configurer when the specified configuration binary file was created. RA and RO modes are mutually exclusive.

Program configured for running in RO mode command line is RA mode

Wrong mode specified, or incorrect option given to the configurer when the specified configuration binary file was created. RA and RO modes are mutually exclusive.

Require at least *<n>* bytes at the top of memory for bootstrapping on processor *<n>*

The bootstrapping sequence requires an extra *<n>* bytes at the top of memory. Once the bootstrapping has finished this memory is available to the user.

Rom size already specified

ROM size must be specified once only.

Rom size in input file and command line do not match

The ROM size specified on the command line does not match that specified to the configurer when the input file was created.

Rom size not specified

A ROM size must be specified because the input file is to be loaded into ROM.

Rom size string invalid

ROM size must be given in decimal.

Rom size string too long

ROM size specified was too large.

Stack size already specified

Stack size must be specified once only.

Stack size string invalid

Stack size must be specified in decimal format.

Stack size string too long

Specified stack size was too large.

Strange function or attribute for linked unit in : *filename*

The collector has found an unfamiliar value in the input file. Either an old version of a tool was used in the creation of the input file, or the input file has been corrupted.

System error

Host system error has occurred, probably when accessing a file. This message may be generated when a file is read and its contents seem to have changed or the file does not exist.

Unexpected end of file : *filename*

One of the files specified in the configuration binary has ended prematurely. *filename* identifies the offending file. If the message 'Suspect corrupted file' is substituted for *filename* then the file is corrupted.

User bootstrap not allowed when program is configured

User defined bootstrap loaders can only be used with single processor programs.

User bootstrap not allowed with rom option

User defined bootstrap loaders cannot be used with ROM-loadable code.

User bootstrap type does not match that of linked unit

Either the target processor type or the error mode of the bootstrap code does not match that of the input file.

3.11.3 Fatal errors**Internal error <message text>**

An internal error has occurred this should be reported to your local INMOS distributor or field applications engineer.

4 `idebug` — network debugger

This chapter is a reference chapter for the network debugger tool `idebug`. It describes the command line syntax and gives examples of the commands to use in different situations. It provides detailed reference information about the debugger symbolic debugging functions and Monitor page commands, and provides a list of error messages.

This chapter does not describe how to use the debugger, which is covered in Chapter 8 of the *Toolset User Guide*.

4.1 Introduction

The network debugger `idebug` is a comprehensive debugging tool for transputer programs. It can be run in *post-mortem* mode to determine the cause of failure in a halted program, or in *interactive* (breakpoint) mode to execute a program stepwise by setting breakpoints in the code. In either mode programs can be debugged from source code using the symbolic functions or from the machine code using the Monitor page commands

Post-mortem debugging allows programs to be examined for the cause of failure after a transputer halts on error. The debugger locates the errant process in the program either by direct examination of the program image in transputer memory or by reading memory dump files. Processes running in parallel with the errant process anywhere on the network can be examined.

Interactive breakpoint debugging allows programs to be executed in a stepwise manner under interactive control. Breakpoints can be set within the code to cause the program to pause for the inspection of variables, channels, and processes; variables can be modified and the program continued with the new values.

The debugger can also be invoked on a dummy network to examine the static features of a program. The dummy network simulates the contents of memory locations and registers, and can also be used to explore the features of the debugger without running a real program.

4.2 Debugging the root transputer

`idebug` can be used to debug single and multitransputer programs. The techniques and commands to use when invoking the debugger differ slightly depending on whether or not the program (or a process forming part of the program) runs on the root transputer, and according to the debugging mode (post-mortem or breakpoint).

Two procedures are used to debug programs in post-mortem mode, depending on whether the application uses the root transputer. Programs that use the root transputer are referred to in this chapter as *R-mode* programs, and programs that do not use the root transputer are referred to as *T-mode* programs. Command line options are used to select the correct mode of operation for `idebug`.

To avoid the need for a memory dump, programs can be *skip loaded* over the root transputer using `iskip`. Skip loading requires at least one extra processor in the network (which will be used by the debugger) but speeds up debugging considerably and is the recommended method where more than one processor is available. Skip loading is described in detail in chapter 15 of this manual.

4.2.1 Board wiring

Before any program can be debugged in post-mortem mode, the transputer's **Analyse** signal must be asserted once, and once only. Because different procedures must be adopted for programs which do and do not use the root transputer, the debugger cannot assert the signal automatically and so the appropriate `iserver` option must be specified on the `idebug` command line.

Table 4.4 summarizes the command sequences to use for the two program modes on different board types.

4.2.2 Post-mortem debugging R-mode programs

Code running on the root transputer, and loaded with `iserver` directly, is debugged in post-mortem mode from a *memory dump* file which is specified by the '**R**' option. The memory dump file must be created using the `idump` tool before the debugger is invoked. Other transputers in the network are debugged down links connected to the root transputer, in the normal way.

For R-mode programs, `idump` asserts the **Analyse** signal and the '**SA**' option is not required on the `idebug` command line. In fact a second assertion of the signal would cause data in the memory to become corrupted. If `idump` is not used before the debugger is run then the debugger cannot load onto the root transputer and a server error is reported.

A description of the `idump` tool can be found in chapter 5 of this manual.

4.2.3 Post-mortem debugging T-mode programs

T-mode programs are loaded using `iskip` and subsequently debugged using the '**T**' option to specify the root transputer link to which the network is connected. The '**SA**' server option must also be added to the `idebug` command line in order to assert **Analyse**.

If the '**SA**' option is not given, the debugger can not be booted onto the root transputer and the server aborts with an error message. The debugger should then be re-invoked with the correct options.

4.2.4 Post-mortem debugging from a network dump file

To suspend a post-mortem R or T debugging session without losing the original context, the Monitor page **N** command can be used to dump the entire state of a network into a *network dump* file. The debugger can then be invoked using this file, without being connected to the network (although one transputer will still be needed to run the debugger).

Note: This option will only work for programs that have not been interactively (breakpoint) debugged.

Memory dump files and network dump files are not the same: the former contains a single processor's memory image while the later contains data about a complete network (they are also in different formats). The *ilist* tool can be used to determine the format of a dump file.

4.2.5 Debugging a dummy network

The debugger may be used to debug a program using dummy data. Using the debugger command line 'D' option which simulates the contents of memory locations and registers, static features of a program may be examined. This is useful to determine processor connectivity and memory mapping for each processor in the network. Because memory locations etc. are simulated, this option only requires the root transputer in order to execute the debugger (even when used with a bootable file for a network of transputers).

The 'D' option may also be used to explore most features of the debugger without running a program.

4.2.6 Methods for interactive breakpoint debugging

Interactive mode breakpoint debugging does not require use of the memory dump tool because the program is automatically skip loaded over the root transputer where the debugger is running. However, like all skip loads it requires an extra processor on the network.

4.3 Running the debugger

The debugger is invoked using the following command line:

► **idebug** *bootablefile* {*options*}

where: *bootablefile* is the bootable file to be debugged.

options is a list of the options given in Table 4.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Only one bootable file may be specified on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax and a list of command line options.

Command line parameters for programs being debugged interactively should *not* be entered on the debugger command line. The debugger will prompt for these parameters when the code being debugged is about to be started.

Note: `idebug` is unique amongst the toolset tools in that, when invoked with command line options, its driver program does not automatically reset (or analyze, as appropriate) the root transputer. This is due to the diversity of hardware configurations where the appropriate sequence may not be obvious to the driver. Because of this, the task of selecting the appropriate `iserver` command is delegated to the user.

Failure to supply the appropriate `iserver` reset (SR) or analyze (SA) options along with `idebug` command line options will result in `iserver` failing to boot `idebug`.

Only when invoked with no command line options at all will `idebug` automatically reset the root transputer and display its own help page.

Option	Description
A	Assert INMOS subsystem Analyse . Directs the debugger to assert Analyse on the sub-network connected to the root processor. Required when using B004 type boards.
AP	A replacement for the A option when running programs on boards from certain vendors. Asserts Analyse on the network connected to the root processor. Contact your supplier to see whether this option is applicable to your hardware. It does not apply to boards manufactured by INMOS.
B <i>linknumber</i>	Interactive breakpoint debug a network that is connected to the root processor via link <i>linknumber</i> . <code>idebug</code> executes on the root processor. Must be accompanied by the <code>iserver</code> 'SR' option.
C <i>type</i>	Specify a processor type (e.g. T425) instead of a class (e.g. TA) for programs that have not been configured.
D	Dummy debugging session. Can be used for familiarization with the debugger or establishing memory mappings. Must be accompanied by the <code>iserver</code> 'SR' option.
GXX	Improves symbolic debugging support for C++ source code. Should be specified when debugging C++ programs.

Option	Description
I	Display debugger version string. Must be accompanied by the iserver 'SR' option.
J #hexdigits	Takes a hexadecimal digit sequence of up to 16 digits and replicates it throughout the data regions of a program (stack, static, heap and vectorspace as appropriate) when interactive debugging. The digit sequence <i>must</i> be preceded by a hash, '#', character. Used when breakpoint debugging configured T426 programs.
K #hexdigits	As the J option but includes freespace. Used when interactive debugging non-configured T426 programs.
M linknumber	Postmortem debug a previous interactive debugging session. idebug executes on the root processor. Must be accompanied by the iserver 'SA' option.
N filename	Postmortem debug a program from a network dump file <i>filename</i> , created by idebug . The file is assumed to have the extension .dmp if none is specified. Must be accompanied by the iserver 'SR' option.
Q variable	Specify environment variable used to specify the ITERM file. The default is " ITERM ".
R filename	Postmortem debug a program that uses the root transputer. <i>filename</i> is the file that contains the contents of the root processor (created by idump or isim). The file is assumed to have the extension .dmp if none is supplied.
S	Ignore subsystem signals when interactive debugging.
T linknumber	Postmortem debug a program that does not use the root processor, on a network that is connected to link <i>linknumber</i> of the root processor. idebug executes on the root processor. Must be accompanied by the iserver 'SA' option.
XQ	Causes the debugger to request confirmation of the Quit command.

Table 4.1 idebug command line options

4.3.1 Toolset file types read by the debugger

The debugger uses information within files produced by toolset tools in order to establish the hierarchy of components used to produce a bootable file.

Table 4.2 provides a list of file types used by the debugger. The table covers all languages which the debugger supports (FORTRAN, C, and occam).

File extension	Description
.f77	FORTTRAN source code file.
.h77	FORTTRAN include file.
.c	C source code file.
.h	C include file.
.occ	occam source code file.
.inc	occam include file.
.cfb	Configuration data file.
.pgm	occam configuration data file.
.btl	Bootable file to be debugged.
.btr	ROM code file to be debugged.
.clu	occam configuration object file.
.lku	Linked unit generated by linker.
.tco	Object file generated by compiler.
.lib	Library file.
.dmp	Root processor dump file (created by idump or isim) or network dump file (created by idebug).

Table 4.2 File types read by debugger

With the exception of a dump file which must have a **.dmp** filename extension, the debugger will accept different extensions for a particular file type. (For example the extensions used by **imakef** such as **.tah** which can be used instead of **.tco**).

4.3.2 Environment variables

idebug requires three environment variables to be set up on the host system (in addition to those required to run the **iserver** and to build a bootable file). These are listed in table 4.3. Details of how to set up these variables can be found in the Delivery Manual that accompanies this release.

ITERM	Contains the name of the file which defines key mappings for debugger symbolic functions and some monitor page commands. The name of the environment variable may be over-ridden by using the 'Q' command line option.
IDEBUGSIZE	Defines the amount of memory available on the root transputer board. This variable must be specified for idebug to work correctly (idebug requires at least 1 Mbyte of available root transputer memory: it is strongly recommended that 2 Mbytes or more be available).
IBOARDSIZE	The amount of memory available for the application program. Required for bootable single transputer programs (created from linked units using <code>icollect</code> with the 'T' option and without the 'M' option), where the memory size was not specified.

Table 4.3 Environment variables used by idebug

4.3.3 Program termination

If the program terminates by issuing the 'terminate' command to the server, the following message is displayed:

```
[Program has finished (after nnn seconds) - hit any key
for monitor page]
```

The debugger can be re-entered after server termination by pressing any key. The final state of the network can be examined using the full range of symbolic and Monitor page commands.

The exit status returned by the program is displayed on the Monitor page.

If the program contains independent processes which require no communication with the server then the debugger allows the program to be resumed. In this case the debugger displays the following warning message:

```
[Warning: iserver terminated by user program: use CTRL-A
for monitor page]
```

4.4 Post-mortem mode invocation

To invoke the post-mortem debugger use the appropriate command from the following list. Command lines are shown in both in UNIX and MS-DOS/VMS formats.

Note: Commands are given for a B008 board wired *subs* (see section 4.7.1). For the commands and command sequences to use on other board types see section 4.7.2.

4.4.1 Debugging T-mode programs – option 'T'

```
idebug bootablefile -t linknumber -sa  
idebug bootablefile /t linknumber /sa
```

where: *bootablefile* is the program bootable file;

linknumber is the number of the link of the root processor which is connected to the network.

Use the 'T' option for programs that do not use the root transputer, that is those loaded by using *iskip*. The program is debugged from the program image that is resident in the memory of each transputer; the information about the rest of the network is extracted down the root transputer link. This method provides the fastest post-mortem debugging because the root transputer memory image is not saved. However, the option does require an extra transputer on the network. The 'T' option should be accompanied by the 'SA' option to assert *Analyse* on the network.

4.4.2 Debugging R-mode programs – option 'R'

```
idebug bootablefile -r dumpfile  
idebug bootablefile /r dumpfile
```

where: *bootablefile* is the program bootable file;

dumpfile is the root transputer memory dump file.

Use the 'R' option for programs that use the root transputer in a network. The dump file is created by using *idump*, which produces a dump of the program image on the root transputer only; the debugger extracts information about other transputers on the network (if applicable) via the root transputer's links.

4.4.3 Debugging a network dump file – option 'N'

```
idebug bootablefile -n netfile -sr  
idebug bootablefile /n netfile /sr
```

where: *bootablefile* is the program bootable file;

netfile is a network dump file.

Use the 'N' option to debug programs without access to the original network of transputers. This is effectively debugging off-line. The network dump file is generated by the *idebug* Monitor page **N** command. Note: this can only be used for programs that have not been debugged interactively. The 'N' option should be accompanied by the *iserver* 'SR' option to reset the network.

4.4.4 Debugging a previous breakpoint session – option 'M'

```
idebug bootablefile -m linknumber -sa  
idebug bootablefile /m linknumber /sa
```

where: *bootablefile* is the program bootable file;

linknumber is the number of the link of the root processor which is connected to the network.

Use the 'M' option to debug a previous breakpoint debugging session where either the network has crashed (error flag was set) or the host `BREAK` key was used to terminate the debugger. This option is the same as the 'T' option but informs the debugger the breakpoint runtime kernel is present. The 'M' option should be accompanied by the *iserver* 'SA' option to assert Analyse on the network. The same action may be achieved when using the debugger in interactive mode with a subsystem wired *subs* (see section 4.7.1) by use of the Monitor page `Y` command.

Note: Symbolic functions and Monitor page commands that support breakpointing are absent in post-mortem mode.

4.4.5 Reinvoking the debugger on single transputer programs

For programs running on a *single transputer only* and debugged from a memory dump file the debugger can be reinvoked on the same dump file by passing the 'SR' option to *iserver* from the *idebug* command line. This option is required to reset the transputer before loading the debugger program (the resetting is normally done by *idump*).

4.4.6 Debugging boot from ROM programs

Programs which are configured to boot from ROM and run in RAM may be debugged in post-mortem mode via a transputer link in a similar manner to that described in section 4.4.1. The debugger must be run on the root processor of the network (as specified to the configurer via the 'P' option) which must be set to boot from link while debugging.

```
idebug romcodefile -t linknumber -sa  
idebug romcodefile /t linknumber /sa
```

where: *romcodefile* is the *.btr* output file produced by *icollect* for use by *ieprom*;

linknumber is the number of the link of the root processor which is connected to the network.

4.5 Interactive mode invocation

To run the debugger in interactive mode use one of the commands below.

Note: Commands are supplied for a B008 board wired *subs*. For the commands to use on other board types see Table 4.4.

```
idebug bootablefile -b linknumber -sr  
idebug bootablefile /b linknumber /sr
```

where: *bootablefile* is the program executable file;

linknumber is the number of the root transputer link where the application network is connected.

In interactive mode *idebug* loads the bootable file directly onto the network and sets up a runtime kernel and *idebug* virtual link system on each processor used by the program. *iserver* is not required to load the program, but an extra processor is required to run the debugger; the program is in effect 'skip' loaded.

When first invoked in interactive mode, the debugger immediately enters the Monitor page where the B (Breakpoint Menu) command can be used to set breakpoints before the program is started.

4.6 Function key mappings

All the debugger symbolic functions, and some Monitor page commands, are assigned to specific keys on the keyboard by the *ITERM* file (the file specified by the environment variable *ITERM*). For the correct keys to use on your terminal consult the keyboard layouts provided in the Delivery Manual that accompanies this release.

ITERM files are supplied with the release for terminals commonly used with your host system but may also be created to suit your own requirements. Details of the *ITERM* file and an example listing which illustrates the format can be found in appendix E.

Key-mapped Monitor page commands are listed in section 4.9.7. A complete list of symbolic functions can be found in section 4.10.

4.7 Debugging programs on INMOS boards

On transputer boards the **Analyse** and **Reset** signals can be propagated from the root transputer in two ways, and this influences the options that must be used when debugging programs.

4.7.1 Subsystem wiring

On transputer boards the subsystem signal is either propagated unchanged to all transputers on the network (known as wired *down*), or the signals are connected to the subsystem port (wired *subs*) from where they are controlled by the board's root processor.

On B004 boards and on all boards where subsystem is wired in the same way **Analyse** must be asserted on the network before transputers can be accessed by the debugger from the root processor. However, if **Analyse** is asserted more than once the program will be corrupted in transputer memory.

The wiring type can be identified by the hardware addresses of the three subsystem registers. On B004-type boards the addresses are as follows:

Signal	Hardware address
Reset	#00000000
Analyse	#00000004
Error	#00000000

An example of a B004-type board is the IMS B404 TRAM. For details of the subsystem wiring on other boards consult the Datasheet or board specification.

In addition, TRAM boards and B004 boards differ in the way the subsystem port is used. On TRAMs these subsystem signals are propagated to all transputers on the network, whereas on B004 boards the signals are not propagated at all.

4.7.2 Debugging options to use with specific board types

The conditions outlined above affect the commands that must be used when debugging T-mode and R-mode programs. table 4.4 shows the command line options to use for different combinations of board type, subsystem wiring, and program mode.

For further details about loading programs see the chapters on loading and debugging in the *Toolset User Guide* (chapters 7 and 8).

4.7.3 Detecting the error flag in interactive mode

In interactive mode the debugger detects that a processor has its error flag set by use of the subsystem services. If the hardware is not wired up to use the subsystem services then the debugger is unable to detect when an error flag is set; this may cause the debugger to hang for no apparent reason. On such networks the **iserver** 'SE' option should be used to detect when an error flag has been set. Note, however, that detection of a set error flag will terminate the debugger without warning — the debugger can, however, be subsequently re-invoked in post-mortem mode.

Note: When using the debugger in interactive mode, the hardware should be set-up to use the subsystem services if possible.

Board	Wiring	Mode	Command line(s) to use
TRAM	down	T	idebug program -b linknumber -sr -se† -s†
			idebug program -m linknumber -sa
			idebug program -t linknumber -sa
		R	idump dumpfile size
			idebug program -r dumpfile
			subs
	idebug program -m linknumber -sa		
	idebug program -t linknumber -sa		
		R	idump dumpfile size
idebug program -r dumpfile			
B004	down	T	idebug program -b linknumber -sr -se† -s†
			idebug program -m linknumber -sa
			idebug program -t linknumber -sa
		R	idump dumpfile size
			idebug program -r dumpfile
			subs
	idebug program -m linknumber -a -sa		
	idebug program -t linknumber -a -sa		
	R	idump dumpfile size	
	idebug program -r dumpfile -a		

Command lines are given in UNIX format ('-' option switch character). For MS-DOS and VMS based toolsets use the '/' option switch character.

The 'SI' option may also be used on any command line to display activity information while the debugger is loading.

Modes: R = program using the root transputer; T = program not using the root transputer, and debugged down a root transputer link.

program is the program bootable file.

† See section 4.7.3.

Table 4.4 Commands to use on different board types

4.8 Debugging programs on non-INMOS boards

If the hardware does not adhere to the INMOS subsystem convention then it is necessary to determine how the hardware is configured with respect to the subsystem and select the appropriate command line options.

It will probably be necessary to use the `idebug` command line 's' option when breakpoint debugging in order to stop the debugger monitoring the subsystem

error status, and the `iserver` 'SE' option to determine when the error flag has been set.

Note: Some non-INMOS boards use a specific subsystem convention which is supported by `idebug`, but which is different from the INMOS convention. To assert subsystem Analyse on such boards, use the 'AP' rather than the 'A' option. The board supplier should be able to say whether the 'AP' option is appropriate for their system.

4.9 Monitor page commands

This section lists and describes the debugger Monitor page commands. The commands are tabulated in alphabetical order for easy reference. Where a command invokes an option submenu the operation of each option in the submenu is described.

Monitor page commands are also listed for easy reference in the Handbook that accompanies this release.

4.9.1 Command format

All Monitor page commands are either single letter commands or are invoked by a single function key press. Key mappings for the few general commands that use function keys can be found in the Delivery Manual that accompanies this release.

4.9.2 Specifying transputer addresses

Many Monitor page commands require a memory address to be entered. Where there is a default value, this is displayed with the prompt. The default address is the last address specified or located to, and is used if RETURN is pressed without entering an address.

Addresses can be specified in decimal or hexadecimal format. Hexadecimal numbers must be given as a sequence of hexadecimal digits preceded by the characters '#', '\$', or '%'. The '#' and '\$' characters are used to prefix a full hexadecimal address. The '%' character adds `MOSTNEG INT` to the hexadecimal value using modulo arithmetic. This is useful when specifying transputer addresses which are signed and start at `MOSTNEG INT`. For example, %70 is interpreted as #80000070 on a 32 bit transputer, and as #8070 on a 16 bit transputer.

Address may also be specified relative to the `Iptr` or `Wptr` (derived from the current `Wdesc`) currently displayed in the monitor page. One of the following forms may be used:

`i+nnn` or `i-nnn`: for addresses relative to `Iptr` — in this case *nnn* is a byte offset.




`w+nnn` or `w-nnn`: for addresses relative to `Wptr` — in this case *nnn* is a word offset.


4.9.3 Scrolling the display

Several commands mapped by the `ITERM` file (see below) may be used to scroll certain of the Monitor page displays. Cursor keys may also be used.

4.9.4 Editing functions

The following string editing functions are available for on-screen editing of strings for certain commands:

START OF LINE	Move the cursor to the beginning of the string.
END OF LINE	Move the cursor to the end of the string.
DELETE LINE	Delete the string.
	Move the cursor left one character.
	Move the cursor right one character.
	Replace the current string with the string used in the previous invocation of the command.
DELETE	Delete the character to the left of the cursor.
RETURN	Enter the string.

Note: **START OF LINE**, **END OF LINE**, **DELETE LINE**, and **DELETE** are mapped by the ITERM file to specific keys on the keyboard. Details of the key mappings on your terminal can be found in the Delivery Manual that accompanies this release.  will not be applicable to some commands.

4.9.5 Commands mapped by ITERM

Certain Monitor page commands are mapped to specific keys on the terminal by the ITERM file. Commands mapped in this way include keys which are used to scroll the display (see below), commands which produce the same effect in both debugging modes, and the commands **TOP**, **RETRACE**, **RELOCATE** and **RESUME** which invoke the corresponding symbolic mode functions.

The keys to use for all Monitor page commands mapped by ITERM can be found by consulting the keyboard layouts supplied in the Delivery Manual.

4.9.6 Summary of commands

Key	Meaning	Description
A*	ASCII	View a region of memory in ASCII.
B†*	Breakpoint	Display the Breakpoint menu enabling breakpoints to be set, cleared or listed.
† = Interactive mode only.		
* = String editing functions available for these commands, see section 4.9.4.		

Key	Meaning	Description
C	Compare	Compare the code on the network with the code that should be there to ensure that the code has not been corrupted.
D*	Disassemble	Display the transputer instructions at a specified area of memory.
E	Next Error	Switch the current display information to that of the next processor in the network which has halted with its error flag set.
F*	Select file	Select a source file for symbolic display using the filename of the object file produced for it.
G	Goto process	Goto symbolic debugging for a particular process.
H*	Hex	View a region of memory in hexadecimal.
I*	Inspect	View a region of memory in a symbolic type. Types are expressed as standard occam types.
J†*	Jump	Start or resume the application program.
K	Processor names	Display the names and types of all processors in the network.
L	Links	Display instruction pointers and process descriptors for the processes currently waiting for input or output on a transputer link, or for a signal on the Event pin.
M	Memory map	Display the memory map of the current processor.
N*	Network dump	Copy the entire state of the transputer network into a 'network dump' file in order to allow continued (off-line) debugging at a later date.
O*	Specify process	Resume the source level symbolic features of the debugger for a particular process.
P*	Processor	Switch the current display information to that of another processor.
Q	Quit	Leave the debugger and return to the host operating system.
R	Run queues	Display instruction pointers and process descriptors of the processes on either the high or low priority active process queue.
S†	Show messages	Display the Messages menu enabling the default actions of the debugger to debug support functions to be changed.
† = Interactive mode only.		
* = String editing functions available for these commands, see section 4.9.4.		

Key	Meaning	Description
T	Timer queues	Display instruction pointers, the process descriptors and the wake-up times of the processes on either the high or low priority timer queue.
U†	Update	Update the monitor page display to reflect the current state of the processor.
V	Process names	Display the memory map of processes on the current processor.
W†*	Write	Write to any portion of memory in a symbolic type. Types are expressed as standard occam types.
X	Exit	Return to symbolic mode.
Y†	Postmortem	Change an interactive breakpoint debugging session into a post-mortem debug session.
Z	Virtual links	Display instruction pointers and process descriptors for processes waiting on the configurator's software virtual links.
?	Help	Display help information.
† = Interactive mode only.		
* = String editing functions available for these commands, see section 4.9.4.		





4.9.7 Symbolic-type commands

HELP	Display help information.
REFRESH	Re-draw the screen.
RELOCATE	Switch to symbolic mode and perform symbolic relocate.
RESUME	Restart a process stopped at a breakpoint.
RETRACE	Switch to symbolic mode and perform symbolic retrace.
TOP	Locate to the last instruction executed on the current processor.

Key bindings for these commands on specific terminal types can be found in the rear of the Delivery Manual.

4.9.8 Scroll keys

LINE UP	Scroll the currently displayed memory, disassembly, queue, or list.
---------	---

LINE DOWN	Scroll the currently displayed memory, disassembly, queue, or list.
PAGE UP	Scroll the currently displayed memory, disassembly, queue, or list.
PAGE DOWN	Scroll the currently displayed memory, disassembly, queue, or list.
TOP OF FILE	Go to the top of the currently displayed processor name list, or software virtual link list.
END OF FILE	Go to the end of the currently displayed processor name list, or software virtual link list.
	Scroll the currently displayed memory, disassembly, queue, or list.
	Scroll the currently displayed memory, disassembly, queue, or list.
	Scroll the currently displayed processor left.
	Scroll the currently displayed processor right.

Key bindings for these commands on specific terminal types can be found in the rear of the Delivery Manual.

4.9.9 Monitor page command descriptions

A ASCII

This command displays a segment of transputer memory in ASCII format, starting at a specific address. If no address is given the last specified address is used. Specify a start address after the prompt:

Start address (#hhhhhhhh) ?

Either press **RETURN** to accept the default (last specified) address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

The memory is displayed in blocks of 16 rows of 32 ASCII bytes, each row preceded by an absolute address in hexadecimal. Bytes are ordered from left to right in each row. Unprintable characters are substituted by a full stop.

The scroll keys (section 4.9.7) can be used to scroll the display.

B Breakpoint menu (interactive mode only)

This command invokes the Breakpoint Menu:

Breakpoint Menu

S - Set a breakpoint on this processor
T - Toggle a breakpoint on this processor
C - Clear a numbered breakpoint
A - Clear all breakpoints on all processors
B - Clear all breakpoints on this processor
E - Set a breakpoint at all entries this processor
G - Set a breakpoint at all entries all processors
M - Set a breakpoint at all main () this processor
L - List all breakpoints and hits
P - List all breakpoints and hits on this processor
Q - Quit this option

Breakpoint option (A,B,C,E,G,L,M,P,Q,S,T) ?

Options are selected by entering one of the single letter commands. A selected option can be cancelled by pressing RETURN with no typed input when it prompts for an address or a breakpoint number.

The 'E' and 'G' options, which set breakpoints at the entry point of all configuration level processes, are mainly for use with OCCAM programs where the entry point of the program is the start of the top-level process. For non-configured occam programs the entry point is the first procedure called when the program starts. For configured occam programs the entry point is the configuration level code.

The 'M' option is intended for languages which have run-time support and a known entry point to the application code (such as `main()` in C, or the FORTRAN main program). This option sets a breakpoint at all the program entry points on the current processor.

Note: for processors which do not have hardware breakpoint support the debugger will not set breakpoints in high priority configuration level processes when the 'E', 'G' or 'M' options are used.

Breakpoints are assigned a unique number which must be used with the 'c' option. Numbers are given on the List Breakpoints displays.

In addition, the List Breakpoint displays provide information about the processor the breakpoint has been placed on (Proc:), the address of the breakpoint (Addr:), the number of times the breakpoint has occurred (Hits:) and for breakpoints set in symbolic mode the filename and line number they correspond to. For example:

```

4) Proc:  0, Addr: #8000408E, "facs.c":201
      Hits: 3
  
```


This display means that breakpoint number 4 on processor 0 at address #8000408E (which corresponds to line number 201 of the file "facts.c") has been hit three times.

Note: Only breakpoints which are set in symbolic mode (at the beginning of a statement) are properly supported. Setting breakpoints at arbitrary addresses using the 's' option may cause incorrect execution of the program.

Note: Breakpoints should not be set in high priority processes on processors without hardware breakpoint support (M212, T212, T222, T414, and T800). The E, G and M options will not set a breakpoint in a high priority (configuration level) process on these processors.

C

Compare memory

This option compares the code actually in memory on the network with the code that was loaded, to check that memory has not become corrupted.

Note: This option treats breakpoints as corrupted code.

The following menu is displayed:

```

          Compare memory
Number of processors in network is : n

A - Check all network processors for discrepancies
B - Check this processor for discrepancies
C - Compare memory of this processor on screen
D - Find first error on this processor
Q - Quit this option

```

Compare memory option (A,B,C,D,Q) ?

Type one of the options A, B, C, D, or Q. Option 'Q' returns to the Monitor page.

Checking the whole network - option A

Option 'A' checks all the processors in the network and displays a summary of the discrepancies found.

If there are no errors the following message is displayed:

```

Checked all processorrrrs in network OK

```

If any errors are detected the number of errors is given along with the address of the first error found and the name of the processor on which it occurred.

Checking a single processor - option B

Option 'B' checks just the current processor. In all other respects it is similar to option 'A'.

Compare memory on screen - option C

Option 'C' displays the actual and expected code for each address in the program code area of the current processor. Discrepancies are marked with an asterisk (*).

Memory is checked in blocks of 128 bytes. At the end of each block, type either 'Q' to quit, or SPACE to read and display the next block.

The format of the display is similar to the following example:

	Processor Code	Correct Code
#800001234 :	0011223344556677	7766554433221100 *
#80000123C :	0011223344556677	0011223344556677
#800001244 :	0011223344556677	7766554433221100 *
...
#8000012AC :	AABBCCDDEff0011	AABBCCDDEff0011

Press [DOWN] to scroll memory, [SPACE] for next error, or Q to quit :

Pressing SPACE automatically invokes option 'D' - Find first error... (see below).

Find first error - option D

Option 'D' searches the current processor's memory for the first occurrence of a discrepancy. If a discrepancy is found the display is the same as in option 'C' above, and the memory can be checked and displayed as in 'Compare memory on screen'.

D Disassemble memory

The Disassemble command disassembles memory into transputer instructions. Specify an address at which to start disassembly after the prompt:

Start address (#hhhhhhh) ?

Either press RETURN to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

The memory is displayed in batches of sixteen transputer instructions, starting with the instruction at the specified address. If the specified

address is within an instruction, the disassembly begins at the start of that instruction. Where the preceding code is data ending with a transputer 'pfix' or 'nfix' instruction, disassembly begins at the start of the pfix or nfix code.

Each instruction is displayed on a single line preceded by the address corresponding to the first byte of the instruction. The disassembly is a direct translation of memory contents into instructions; it neither inserts labels, nor provides symbolic operands.

The scroll keys (section 4.9.7) can be used to scroll the display.

E Next Error

Next Error searches forward through the network for the next processor which has both its error and halt-on-error flags set. Processors are searched in the same order as they are listed by the 'K' command, starting from the current processor and wrapping round. If a processor is found with both flags set the display is changed to the new processor as if the 'P' option had been used. Press **TOP** to display the source line which caused the error.

If there is only one processor in the network then a message to this effect is displayed.

F Select source file

This command enables a program source file to be displayed within the symbolic debugging environment for a particular processor. This allows breakpoints to be set in modules which have not yet been reached in the program's execution. (Source which has not yet been executed cannot be displayed using the 'O' or 'G' options because the `Iptr` and `Wdesc` addresses are not yet known.) This command may also be used to browse source files rather like the **CHANGE FILE** symbolic function. However, unlike **CHANGE FILE** it allows some of the symbolic debugging operations to be used.

Note: Editing keys may be used with this command to provide a simple *history* mechanism (see section 4.9.4).

For mixed language programs, the behavior of this command will differ depending on whether `iconf` or the Occam configurer `occonf` has been used to configure the program. (`iconf` is supplied with FORTRAN and ANSI C toolsets, `occonf` with Occam toolsets.)

The differences in the behavior are described below:

Behavior of command when no configurer or icconf is used

If a processor has been configured to contain more than one process, this option first prompts for the process number corresponding to the source code:

Select process number (0 - N) ?

The range of numbers displayed in brackets are process numbers assigned by the debugger to different processes on the processor. The process numbers assigned to process names by the debugger can be determined by using the Monitor page Process Name ('V') option before invoking the 'F' command.

Once a valid process number has been supplied (if applicable), the debugger prompts for the filename of the *compiled* object module. The full object filename (including extension — conventionally .tco) must be supplied.

Object module filename ?

The object filename must be specified because the debugger extracts the source code filename from the debug information in the compiled object file.

Note: Editing keys may be used with this command to provide a simple history mechanism (see section 4.9.4). At each prompt this command may be aborted by pressing RETURN with no typed input.

Behavior of command when oconf is used

The debugger first prompts for the filename of a *linked* object module. The full linked filename (including extension — conventionally .lku) must be supplied.

Linked unit filename ?

The linked filename must be specified because the debugger needs to know which linked unit (incorporated by a configurer #USE directive) contains the object code from the source file to be displayed.

The debugger then prompts for the filename of a *compiled* object module contained within the selected linked unit. The full object filename (including extension — conventionally .tco) must be supplied.

Object module filename ?

The object module filename must be specified because the debugger extracts the source code filename from the debug information in the compiled object file.

At each prompt this command may be aborted by pressing **RETURN** with no typed input.

G Go to process

This command locates to the source code for any process which is currently shown on the screen. Any process displayed by using the **I**, **L**, **R**, **T**, or **Z** commands may be selected.

The cursor is positioned next to the **Ip_{tr}** value and permitted responses are listed on the screen, as follows:

Goto process – use **CURSOR** then **RETURN**, or 0 to **F**, **(I)p_{tr}**, **(L)o** or **(Q)uit**

To select a process and display the source code, move the cursor to a displayed **Ip_{tr}** value and press **RETURN**, or use one of the following short-cuts:

- The **I** option locates to the current process (the process corresponding to the displayed **Ip_{tr}** and **Wdesc** values).
- If currently in high priority, the **L** option can be used to locate to the interrupted low priority process (the process corresponding to the displayed **Ip_{tr}IntSave** and **WdescIntSave** values).
- The hex numbers **0** — **F** will locate to the process corresponding to one of the 16 lines displayed on the right hand side of the Monitor page (the entries in the timer or run queues, or processes waiting on links).

Type **'Q'**, **FINISH**, or **REFRESH** to abort this command.

H Hex

The Hex command displays memory in hexadecimal. Specify the start address after the prompt:

Start address (#hhhhhhhh) ?

Press **RETURN** to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by **#**, or the short form **'%h...h'**. If the specified start address is within a word, the start address is aligned to the start of that word.

The memory is displayed as rows of words in hexadecimal format. Each row contains four or eight words, depending on transputer word length. Words are displayed in hexadecimal (four or eight hexadecimal digits depending on word length), most significant byte first.

For a four byte per word processor the sequence of bytes in a single row would be:

3 2 1 0 7 6 5 4 11 10 9 8 15 14 13 12

For a two byte per word processor, the ordering would be:

1 0 3 2 5 4 7 6 9 8 11 10 13 12 15 14

Words are ordered left to right in the row starting from the lowest address. The word specified by the start address is the top leftmost word of the display.

The address at the start of each line is an absolute address displayed in hexadecimal format.

The scroll keys (section 4.9.7) can be used to scroll the display.

I Inspect memory

The Inspect command can be used to inspect the contents of an entire array. Specify a start address after the prompt:

Start address (#hhhhhhhh) ?

Either press **RETURN** to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

When a start address has been given, the following prompt is displayed:

Typed Memory Dump

0 - ASCII
1 - INT
2 - BYTE
3 - BOOL
4 - INT16
5 - INT32
6 - INT64
7 - REAL32
8 - REAL64
9 - CHAN
Q - Quit this option

Which occam type (1 - INT) ?

Give the number corresponding to the type of data to be displayed, press **RETURN** to accept the default type or enter Q to quit this option.

The types correspond to formal OCCAM types as defined in the *occam 2 Reference Manual*. OCCAM equivalences of C and FORTRAN types are listed in table 4.5.

C	FORTTRAN	occam
int	INTEGER	INT
char unsigned char	CHARACTER	BYTE
	LOGICAL	BOOL
short signed short	INTEGER*2	INT16
long signed long	INTEGER*4	INT32
float	REAL REAL*4	REAL32
double long double	DOUBLE PRECISION REAL*8	REAL64

Table 4.5 Type equivalents for Inspect command

ASCII arrays are displayed in the format used by the Monitor page command **[A]**. Other types are displayed both in their normal representation and hexadecimal format.

The memory is displayed as sixteen rows of data. The address at the start of each line is an absolute address displayed as a hexadecimal number. The element specified by the start address is on the top row of the display.

Start addresses are aligned to the nearest valid boundary for the type, that is: BYTE and BOOL to the nearest byte; INT16 to the nearest even byte; INT, INT32, INT64, REAL32, REAL64, and CHAN to the nearest word.

The scroll keys (section 4.9.7) can be used to scroll the display.

J**Jump into and run program**

This command starts up a program from the Monitor page, or restarts a process which has encountered a breakpoint or has been stopped by one of the debug support functions (for details of these functions see the appropriate *Language and libraries* manual).

Starting a program

When starting a program the debugger converts (*patches*) the configuration external channels (those assigned to links) for each processor into *virtual* channels for use with the debugging kernel. This action is indicated by an activity indicator.

When the patching is complete the debugger prompts for a command line for the program:

Command line:

Simply press **RETURN** if the program does not require any command line parameters. The program will then start running.

Resuming from a breakpoint

When jumping into and resuming a program from a breakpoint, in the monitor page, the following menu is displayed:

Jump into Application

- R - Resume process stopped at a breakpoint
- O - Resume monitoring of network
(abandon process stopped at a breakpoint)
- J - Restart process at a different location
- Q - Quit this option

Which option (J,O,Q,R) ?

Note: the **RESUME** key can be used as an alternative to using this command with the R option.

Resuming from debug support functions

When resuming from one of the debug support functions, the following sub-menu is displayed:

Jump into Application

- O - Resume monitoring of network
(abandon process stopped at a program error)
- J - Restart process at a different location
- Q - Quit this option

Which option (J,O,Q) ?

Jump options

The four Jump options are described in the following table:

Option	Description
R	Restarts the process that encountered the breakpoint.
O	<p>Ignores the stopped process and resumes monitoring the network for other process activity.</p> <p>Note: When a process has stopped, other processes continue to run until they either encounter a breakpoint or program error (i.e. one of the debugging support functions), or become dependent on the stopped process.</p> <p>Note: Using this option for a process stopped on a breakpoint removes the process forever.</p>
J	Restarts the process from a different location. Only use this option if you are confident that the program can be resumed from the new location; resumption from most locations will corrupt the program.
Q	Quits the Jump sub-menu.

Editing keys

Editing keys may be used with this command when entering the command line parameters starting the program, see section 4.9.4.

K

Processor names

This command displays the internal processor numbers corresponding to processor names used in the configuration description and the corresponding processor type. Processor *numbers* must be given when selecting specific processors for display.

The scroll keys (section 4.9.7) can be used to scroll the display.

Note: The debugger displays only the first 19 characters of the processor name. If this is a problem then names should be made unique within the first 19 characters.

L

Links

The Links command displays the instruction pointer, workspace descriptor, and priority of the processes waiting for communication on the links, or for a signal on the Event pin. If no process is waiting, the link is described as 'Empty'. Link connections on the processor, and the link from which the processor was booted, are also displayed.

If a processor uses software virtual links then the title line of the data displayed appears as:

Link Information (virtual links present)

This is to warn that the link state information may be for software virtual processes which cannot be located to in the normal manner. In this case it is more useful to use the Monitor page **Z** command to display the software virtual links instead.

For configured programs, the debugger checks that the link the root processor has been booted from matches that expected by the configurer. If it does not, the following message is displayed:

```
Booted from link n < Should be link m !!! >
```

The format of the display is similar to the following:

```

Link Information
Link 0 out Empty
Link 1 out Empty
Link 2 out Iptr: #80000256 Wdesc: #80000091 (Lo)
Link 3 out Empty
Link 0 in Empty
Link 1 in Empty
Link 2 in Iptr: #80000321 Wdesc: #80000125 (Lo)
Link 3 in Iptr: #80000554 Wdesc: #80000170 (Hi)
Event in Empty

Link 0 connected to Host
Link 1 not connected
Link 2 connected to Processor 1, Link 1
Link 3 connected to Edge
```

```
Booted from link 0
```

M

Memory map

The Memory map command displays the memory map of the current processor along with the mode that `idebug` is currently in. The mode may be one of:

Mode	Description
Interactive Mode	Interactive breakpoint debugging session
Postmortem Interactive Mode	Postmortem debugging session of a previous interactive debugging session.
Postmortem Mode	Postmortem debugging session either down a transputer link or from a dump file.
Dummy Session	Dummy debugging session

The display includes the address ranges of on-chip RAM, program code, configuration code, stack (workspace) and vectorspace, the sizes of each component in bytes rounded up to the nearest 1K bytes, total memory usage, and the address of MemStart (the first free location after the RAM reserved for the processor's own use).

Also displayed is the total memory usage. Total memory usage indicates the amount of memory used by a user program. Note that this may include a region of memory at the beginning of freespace; this area is used for configuration code before execution of a user program starts (this memory may be safely overwritten by the user program because the configuration code finishes executing before the user code starts).

The minimum memory size specified to the configurer or the collector, or as defined in `IBOARDSIZE` as appropriate, must be at least as large as the total memory usage for each processor.

Also displayed is the maximum number of processors that can be accommodated by the debugger's buffer space. This will depend on the amount of memory on the root processor, indicated to the debugger by the host environment variable `DEBUGSIZE`.

The address of MemStart is the value actually found on the transputer in the network. If this does not correspond to that expected by the configuration description, for example if a T414 was found when a T805 was expected, the following message is displayed:

```
MemStart:           #80000048 (T414) ?
MemStart should be: #80000070 (T805)    <<!>>
```

If an incorrect MemStart is detected the symbolic functions may not work correctly. In this case the program should be rebuilt for the correct processor types before reinvoking the debugger.

N

Network dump

The Network dump command saves the state of the transputer network for later analysis. If the debugger is terminated without creating a network dump file, debugging cannot continue from the same point without re-running the program. This is because the debugger itself overwrites parts of the memory on each transputer in the network.

Note: This command cannot be used in interactive mode (`idebug` command line option 'B') or when post-mortem debugging a breakpoint session (`idebug` command line option 'M').

Once a network dump file has been created, debugging can continue from the dump file, and the debugger does not need to be connected to the target network.

Before the dump file is created, the debugger calculates the disk space required, and requests confirmation. The size of the file depends on how much of each processor's memory is actually used in running the program, and is displayed as follows:

```

      Create network dump file
Number of processors to dump : 2
File size excluding Freespace : 112604 bytes
File size including Freespace : 2097308 bytes

```

Continue with network dump (Y,N) ?

To continue with the network dump, type 'Y'.

The debugger will then ask whether to include freespace in the dump file (this is not normally required for configured programs).

Do you wish to include Freespace (Y,N) ?

Type 'Y' or 'N' as appropriate and specify a filename after the prompt:

Filename ("network.dmp" or "QUIT") ?

Press to accept the default filename, enter a filename (any extension will be replaced by '.dmp'), or type 'QUIT' (uppercase) to exit.

If the file already exists, confirmation is asked for:

```

File "network.dmp" already exists
Overwrite it (Y,N) ?

```

If 'N' is entered then a new filename is asked for.

While the dump file is being written, a message is displayed at the terminal. For example:

```

Dumping network to file "network.dmp" ...
Processor 1 (T800)
Memory to dump : 10456 bytes ...

```

☐ Specify process

This command returns to symbolic debugging, either at the same source line, or at another location. It can be used to locate to any source line, whether or not a process is waiting or executing there. To ensure the debugger locates to a valid process, it is better to use the 'G' command.

To return to symbolic debugging, the debugger requires values for `Iptr` and `Wdesc`. Specify `Iptr` after the prompt:

`Iptr` (#hhhhhhhh) ?

The default displayed in parentheses is the last line located to on this processor, or the address of the last instruction executed. Either press **RETURN** to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

Useful process addresses can be determined using the **L**, **R**, or **T** commands to display processes. These processes can, however, be more easily located by using the **G** command. The value of the saved low priority **Iptr** can also be used.

If the **Iptr** is not within the program body, the debugger indicates the type of code to which it corresponds.

After the **Iptr** has been entered the debugger prompts for the value of the process descriptor:

Wdesc (#hhhhhhhh) ?

If a displayed **Iptr** was specified, its corresponding **Wdesc** is offered as a default. Press **RETURN** to accept the default, or specify a value in the same format as the **Iptr**.

If no symbolic features other than a single 'locate' are required, then **Wdesc** is not needed and the default can be accepted.

If an invalid **Wdesc** is given, most of the symbolic features will not work, or will display incorrect values. However, the values of scalar constants and some other symbols can still be determined.

Any attempt to inspect or modify variables or channels, or to backtrace, will give one of the following messages:

Wdesc is invalid - Cannot backtrace
Wdesc is invalid - Cannot Inspect variables
Wdesc is invalid - Cannot Modify variables
Wdesc is invalid - Cannot auto backtrace out of library

Once the **Iptr** and **Wdesc** have been supplied, the debugger displays the source code at the required location, and the full range of symbolic features are available.

P Change processor

This command changes to a different processor in the network. Specify the processor number after the prompt:

New processor number (0 - n) ?

To determine the mapping between the processor number and the processor name used in the configuration file, use the 'K' command. If the proces-

sor exists the display is changed to provide information about the specified processor. If the new processor's word length is different from that of the previous processor, the start address is reset to the bottom of memory. If the processor is not in the configuration, the following message is displayed:

Error : That processor number does not exist

To abort the command press **RETURN** with no input.

If there is only one processor in the network, an appropriate message is displayed.

The scroll keys (section 4.9.7) can be used to change the displayed processor. The sequence of processors is the same as that displayed by the 'K' command.

Q Quit

This command quits the debugger and returns to the operating system. Once quit, the debugger cannot be used to debug the same program without reloading the program unless a 'network dump' file has been created. This is because using the debugger overwrites some of the contents of the network.

If the command line option **XQ** has been used the debugger will ask for confirmation before quitting.

R Run queues

This command displays **Iptrs** and **Wdescs** for processes waiting on the processor's active process queues. If both high and low priority front process queues are empty, the following message is displayed:

Both process queues are empty

If neither queue is empty, the debugger asks which queue is to be displayed:

High or low priority process queue ? (H,L)

Type 'H' or 'L' as required. If only one queue is empty, the debugger displays the non-empty queue.

The screen display is paged. The scroll keys (section 4.9.7) can be used to scroll the display.

Note: In interactive mode this command may show the details of a process more than once. The string '<!>' next to the queue heading serves as a reminder that this may occur.

Processes which belong to the debugging kernel are also displayed and identified with the string '(Runtime kernel)'.

[S] Show debugging messages

This command is used to enable and disable debugging messages and prompts. It invokes the following submenu:

Show Messages Menu

```
B -- Show message for breakpoints      : ON
D -- Show debug messages               : ON
E -- Show message for program errors   : ON
Q -- Quit this option
```

which option (B,D,E,Q) ?

Options B and E control the display of prompts when a breakpoint or program error (i.e. one of the debug support library functions `debug_assert()` and `debug_stop()`, etc.) is encountered. Disabling these options ensures that the debugger is entered on a breakpoint or error without requesting confirmation.

Option D controls the display of debugging messages inserted with the library functions `debug_message()`, etc.

[T] Timer queues

This command displays `Iptrs`, `Wdescs`, and wake-up times for processes waiting on the processor's timer queues. Prompts and displays are similar to those for the Run queues command.

[U] Update registers

This command updates the clock and status display (e.g. run queues, links) for the current processor. It enables the activity of other processes to be monitored while one process is stopped at a breakpoint or error.

[V] Process memory map

This command provides details on each process resident on a processor. This consists of user processes, and configuration processes for virtual channels (if applicable: their names begin with a '%'); it does not include the debugging kernel used by the debugger when interactive breakpoint debugging (this information is shown by the Memory Map option).

A process is assigned a number by the debugger in order to identify the process when using certain other monitor page options. In addition to the process number, the following details are provided: the name of the process, the priority the process starts in, and the process stack and code areas. Note that for non-configured C programs, the stack area displayed is not that used by the program (the actual stack used is provided by the free-space area).

An example output for two configured processes is shown below:

```

      Process Memory Map
Process  0 : "occam_process"
  Initially : High priority
    Stack : #80005000 - #80005023
    Code  : #8000A038 - #8000A04B

Process  1 : "c_process"
  Initially : Low priority
    Stack : #80005024 - #8000A037
    Code  : #8000A04C - #8000D73B

```

Note: The debugger displays only the first 19 characters of the process name. If this is a problem then names should be made unique within the first 19 characters.

Note: Processes placed on a processor to provide software virtual links have names starting with '%'.

W Write to memory

This command writes a value to a specified address. Values must be specified in the current type (the type used in the previous Monitor page Inspect command), or the Occam type **INT** if the type was a **CHAN** or the Disassemble or Hex options have been used after an Inspect.

X Exit

This command returns to symbolic mode and locates to the current address.

Y Enter post-mortem debugging

This command allows the debugger to be switched into post-mortem mode when the program crashes (a process sets the error flag on any processor). Halted processors prevent the breakpoint debugger from accessing the network correctly and debugging must continue in post-mortem mode. It has the same effect as re-invoking the debugger with the command line '**M**' option.

If the program has not already started, then the debugger prompts for confirmation:

The program has not started - are you sure (Y,N) ?

If the program has not crashed, the debugger prompts for confirmation:

The program has not crashed - are you sure (Y,N) ?

If checking of the subsystem error status is disabled (with the command line '**S**' option), then the prompt is:

Unable to detect if the program has crashed - are you sure (Y,N) ?

Typing 'Y' continues the operation, typing 'N' aborts it.

This command will only work if the subsystem is wired *subs* (see section 4.7.1). For a subsystem wired *down*, it is necessary to quit and restart the debugger using the Monitor page 'M' command line option (instead of the previous breakpoint command line 'B' option).

Note: State information for a process that has stopped (on breakpoint or error) will be lost when switching from breakpoint to post-mortem mode. If the information is important it should be noted before switching modes.

Z Display software virtual links

This command displays instruction pointers and process descriptors for the processes currently waiting on software virtual links placed onto a processor by the configurer. All of the virtual output links (displayed as `Vout N`) are displayed followed by all of the virtual input links (displayed as `Vin N`).

The scroll keys (section 4.9.7) can be used to scroll the display.

In order to establish the mapping of user channels onto software virtual links on a particular processor, you should use the configurer information 'I' option when configuring.

Note: all low priority user processes using software virtual links will be promoted temporarily to high priority when they communicate. The debugger cannot tell if they were originally at high or at low priority. If you need to specify a low priority `Wdesc` then use the displayed value with the least significant bit set (e.g. `%1234` becomes `%1235`).

4.9.10 Symbolic-type commands

TOP

This command is used to display the source corresponding to the last instruction to be executed on the current processor. It is the same as typing 'G', then 'I' (or 'G', then **RETURN**).

RELOCATE

This command returns to symbolic mode and performs a symbolic **RELOCATE**. It cannot be used if the processor has been changed at the Monitor page.

RETRACE

This command returns to symbolic mode and performs a symbolic **RETRACE**. It cannot be used if the processor has been changed at the Monitor page.

RESUME

This command resumes a process stopped at a breakpoint in a similar manner using the **RESUME** command when in symbolic mode. It is a shorthand equivalent to using the J command and selecting the R option to resume a process stopped at a breakpoint from the monitor page.

?

HELP

These commands display a summary of the commands available at the Monitor page.

REFRESH

This command refreshes the screen.

4.10 Symbolic functions

Symbolic debugging allows high level language programs to be debugged from the identifiers used in the source code. Symbolic identifiers are the names given in the program to variables, constants, channels, and functions.

Symbolic functions are invoked using keyboard function keys. Keyboard layouts that show the key bindings for common terminal types can be found in the Delivery Manual that accompanies the release. The symbolic functions are summarized alphabetically below. Each description includes a reference to the page where the command function is described in more detail.

Note: † = Functions only available in interactive mode.

BACKTRACE

Locate to the calling function or procedure [page 146].

END OF FILE

Go to the last line in the file [page 147].

CHANGE FILE

Display a different source file [page 147].

CHANNEL

Locate to the process waiting on a channel [page 144].

CONTINUE FROM †

Restart a stopped process from the current line [page 145].

ENTER FILE

Change to an included source file [page 147].

EXIT FILE

Return to the enclosing source file [page 147].

FINISH

Quit the debugger [page 148].

GET ADDRESS

Display the location of a source line in memory [page 147].

GOTO LINE	Go to a specific line in the file [page 146].
HELP	Display a summary of commonly used symbolic functions [page 147].
INFO	Display process information (e.g. instruction pointer, process descriptor, process name) [page 144].
INSPECT	Display the type and value of a source code symbol [page 143].
INTERRUPT †	Force the debugger into the Monitor page without stopping the program [page 145].
MODIFY †	Change the value of a variable in memory [page 145].
MONITOR	Change to the monitor page [page 148].
RELOCATE	Locate back to the last location line [page 146].
RESUME †	Resume a process stopped at a breakpoint [page 145].
RETRACE	Undo a BACKTRACE [page 146].
SEARCH	Search for a specified string [page 147].
TOGGLE BREAK †	Set or clear a breakpoint on the current line [page 145].
TOGGLE HEX	Enables/disables hex-oriented display of constants and variables for C and FORTRAN [page 147].
TOP	Locate back to the error or last source code location [page 146].
TOP OF FILE	Go to the first line in the file [page 147].

4.10.1 Symbolic functions

INSPECT

This function displays the value and type of source code symbols. To inspect a symbol, use the cursor keys to position the cursor on the required symbol and press **INSPECT**. If the cursor is not on a valid symbol when **INSPECT** is pressed, a symbol name is prompted for. Type **RETURN** to abort the **INSPECT** operation, or type a name followed by **RETURN**. The case of letters in a variable name is significant — except when debugging

FORTRAN where case is not significant. Variable names which contain spaces must be entered *without* the spaces. Specifying an empty expression aborts the **INSPECT** operation.

The symbol must be in scope from the line to which the debugger last located, which may not be different from the current cursor position. If the symbol is not in scope at that location, or not found at all, one of the following messages is displayed (depending on the language being debugged):

```
Name 'symbol' not in dynamic scope
Name 'symbol' not found
error: identifier "name" is not in scope
error: identifier "name" not found
```

Expression language

INSPECT supports an expression language for examining source file symbols. Details of the language and display formats for symbols can be found in the appropriate, language specific, sections below.

CHANNEL

This function 'jumps' down a channel if a process is waiting at the other end. This is used in the same way as **INSPECT**, but positioned on a channel. The debugger locates to the source line corresponding to the waiting process; that process can then be debugged.

The **CHANNEL** function will jump to other processors along transputer links as long as the program has not been configured to use virtual links (see section 8.4 of the debugging chapter in the *Toolset User Guide*). If a process running on another processor is waiting for communication on a channel the debugger jumps down the link and automatically changes to that processor.

INFO

This function displays the **Iptr** and **Wdesc** of the last location, the process name and priority, and the processor number.

If the **Wdesc** is not in the defined region for a process the message: **Undefined process** is displayed in place of the process name. For single processor programs that have not been configured there is no defined region and the message: **<Stack area unknown>** is displayed to reflect this.

If a **Wdesc** has not been supplied, it is shown as 'invalid'.

4.10.2 Interactive mode functions

The following functions are only available when running in interactive mode.

TOGGLE BREAK

This function toggles a breakpoint on the source line indicated by the cursor and provides information on the breakpoint number (as used by the Monitor page **B** command), whether it was set or cleared, and the line number it is on.

When the source line the cursor is on produces no associated object code the debugger displays an exclamation mark (<!>) after the line number to indicate that the breakpoint has been toggled on a different line to the one the cursor is on (as shown at the bottom of the display).

RESUME

This function restarts a process stopped at a breakpoint. (To restart a process which has been stopped by one of the debug support functions use **CONTINUE FROM**).

CONTINUE FROM

This function restarts the program from the line indicated by the cursor. **CONTINUE FROM** should only be used to resume a process after it has been stopped by one of the debug support functions. The result of continuing from other points in the code may be unpredictable if there are intervening stack adjustments.

MODIFY

This function changes the value of a variable in transputer memory. For C and FORTRAN, **MODIFY** accepts expressions involving any symbol in scope. To modify a variable place the cursor on the name and press **MODIFY** .

Expression language

Variables to be modified can be specified using the **INSPECT** expression language. Details of the syntax can be found in the following language specific sections.

INTERRUPT

This function forces the debugger to enter the Monitor page without stopping the program.

Note: This command does not operate if there are keystrokes waiting before it in the keyboard buffers. It may also fail if the application program is waiting for input from the keyboard.

Note: A side effect of this command is that the debugger suspends **iserver** communications in order to preserve debugger output to the screen.

4.10.3 Locating functions

The following functions are used to change the debugger's current 'location'. The current location is the point at which the all symbolic functions apply — associated with the location are an instruction pointer value and a workspace address. The initial location will generally be a breakpoint (in interactive mode) or the point where an error occurred (in post-mortem mode).

BACKTRACE

This function locates to the line where a procedure or function was called. If the debugger is already located in the program's topmost procedure, no backtrace is possible.

RETRACE

This function locates back to the previous place where the debugger was located. Repeated use of **RETRACE** reverses the effect of previous **BACKTRACE** operations.

RELOCATE

This function returns the cursor to the last place located to by the debugger. For example, it can be used to return to the original source line of an error after browsing other areas of the code with the cursor keys or the file selection functions.

TOP

This function locates back to the line containing the original breakpoint or error, or to the line located to by the previous use of the monitor page **G** or **O** commands.

4.10.4 Cursor and display control functions

These functions are used to move the cursor around the program and to view other source files.

GOTO LINE

This function locates to a particular line in the source file. Specify a line number, or type **RETURN** to abort the operation.

TOP OF FILE

Moves to the start of the current file.

END OF FILE

Moves to the end of the current file.

SEARCH

This function searches forwards in the source file for a specified string. Either specify a search string or press **RETURN** to accept the default, which is the last string specified.

ENTER FILE

Enters an included source file. Position the cursor on the include directive and press **ENTER FILE**.

EXIT FILE

Exits from an entered include file.

CHANGE FILE

This function opens a different source file for reading only. No symbolic functions are available, unlike the Monitor page 'F' option.

4.10.5 Miscellaneous functions

HELP

This function displays a help screen of the commonly used debugger symbolic functions.

GET ADDRESS

This function displays the address of the transputer code corresponding to the source line where the cursor is currently placed (*not* necessarily the current 'location')

TOGGLE HEX

This function turns the display of hexadecimal values of variables on and off. When enabled, the debugger displays hexadecimal as well as decimal

values. The default for C and FORTRAN is to display variables in decimal format only.

MONITOR

This function switches to the Monitor page environment.

FINISH

This function quits the debugger. The Monitor page 'Q' command has the same effect. If the command line option 'xq' was used then the debugger will ask for confirmation before quitting.

4.11 INSPECT/MODIFY expression language for C

The expression language for source code symbols (variables, constants, and channels) follows the syntax of the C programming language with some minor modifications.

4.11.1 Syntax not supported

Table 4.6 lists the standard C expression syntax *not* supported in the debugger expression language.

Area of limitation	Example
Casting to pointer types	<code>(char *) ptr</code>
Calling of functions	<code>sqrt (x)</code>
Entry of strings	<code>"a string"</code>
Entry of initializer lists	<code>{ 1, 2, 3 }</code>
Increment and decrement operators	<code>++count</code>
Trigraph sequences	<code>'??('</code>
Bit field modification	
Modification by assignment operators	<code>x = y</code> <code>n += 1</code>
Conditional operator	<code>a ? b : c</code>

Table 4.7 Limitations to syntax

In addition, the 'address of' operator '&' returns an `int` rather than a pointer to the type.

4.11.2 Extensions to C syntax

Subarrays

The language supports the specification of array subranges for *arithmetic* data types. Subranges are specified as two array bounds separated by a semicolon. For example: `foo[2;4]` displays the values of elements `foo[2]`, `foo[3]` and `foo[4]`.

Note: For arrays and structures the information displayed will normally overwrite part of the screen display. Press any key, when prompted, to restore the display.

Scope resolution operator

The *scope resolution* operator `::` is available when debugging both C and C++ programs. This allows access to a global identifier which has been hidden by a local declaration, for example:

```
static int foo = 42;

void example(void) {
    int foo = 321;

    debug_stop(); /* program will stop here */
}
```

In this example, when the program has stopped at the `debug_stop()` function, the identifier `foo` can be inspected and the value 321 (the value that is currently in scope) will be displayed. If `::foo` is inspected then the value 42 will be displayed.

Hex constants

The hex constant syntax has been extended to accept a `'%'` character after the leading `0x` component of a hex constant. This provides a shorthand mechanism for specifying transputer addresses in a similar manner to that provided in the Monitor page. The `'%'` character adds `INT_MIN` (the most negative integer) to the hex constant using modulo arithmetic.

For example, `0x%70` produces the value `0x80000070` on a 32 bit transputer and `0x8070` on a 16 bit transputer.

Address constant indirect

When using `INSPECT` or `MODIFY` a constant expression which has type `int` or `unsigned int` may be de-referenced. Normally only pointers may be de-referenced: this addition removes the need to change to the Monitor page to inspect memory locations.

For example, `*0x80000000` (or `*0x%0`) would display the integer at memory location `0x80000000` on a 32 bit processor.

4.11.3 Automatic expression pickup

When `INSPECT` or `MODIFY` is selected, `idebug` will automatically 'capture' the identifier which is underneath the cursor (if any). The captured expression can then be modified (using the editing keys described below) before applying the selected option.

In this release the automatic capture is more eager for simple `struct` or `union` member expressions which contain only the `.` and `->` operators.




This is best illustrated by example. In the following examples, the cursor is positioned over `baz` when `INSPECT` or `MODIFY` is selected:

Program text	Expression captured
baz	baz
baz.ptr	baz.ptr
*(baz).ptr	baz
*baz.ptr	baz.ptr
baz.ptr->ptr	baz.ptr->ptr
baz.foo.ptr	baz.foo.ptr
baz->foo->ptr	baz->foo->ptr
baz[x].ptr	baz

In addition, for those captured expression which match the the program text, the cursor may be positioned anywhere on the expression before selecting **INSPECT** or **MODIFY**.

4.11.4 Editing functions

The following functions are available for the on-screen editing of expressions:

START OF LINE	Move the cursor to the beginning of the expression.
END OF LINE	Move the cursor to the end of the expression.
DELETE LINE	Delete the expression.
	Move the cursor left one character.
	Move the cursor right one character.
	Replace the current expression with the expression used in the previous INSPECT or MODIFY operation.
DELETE	Delete the character to the left of the cursor.
RETURN	Enter the expression for evaluation.

Note:

START OF LINE, **END OF LINE**, **DELETE LINE**, and **DELETE** are mapped by the **ITERM** file to specific keys on the keyboard. Keyboard layouts can be found in the Delivery Manual.

4.11.5 Warnings

When evaluating an expression, checking is performed which may lead to warning messages being produced (e.g. overflow in arithmetic operation, mis-aligned

pointer). Such warnings are intended to highlight potential problems and to ensure that a user understands any action `idebug` is taking.

4.11.6 Types

C types are interpreted and displayed by the debugger as follows:

ANSI C types are categorized, by the debugger, as shown in table 4.8. These categories define the operations that can be performed on the various types and are also used in error messages when invalid operands are used in expression syntax. For example, arithmetic can be performed on any of the 'scalar' types, but attempting to use a 'derived' type, such as a `struct`, in an expression produces an error message of the form "error: non-scalar left-hand operand..."

Name	Member types
character	char, signed char, unsigned char
floating	float, double, long double
basic	character, signed integer, unsigned integer, floating
integral	character, signed integer, unsigned integer, enum
arithmetic	integral, floating
scalar	arithmetic, pointer
derived	array, function pointer, struct, union

Table 4.8 Type categories in C

Type compatibility when using `MODIFY`

Source and destination expressions must be type compatible according to the rules of C. Scalar types are cast automatically into other scalar types but non-scalar expressions must be strictly compatible.

Type conversions, where required, are performed according to normal C promotion rules.

The following examples illustrate the rules governing type compatibility.

Given the following declarations:

```
int  two_d_array[2][10];
int  one_d_array[10];
int  foo;
char bar;
```

Then the following modifications are permitted:

```
Source:    one_d_array          (array of 10 integers)
Destination: two_d_array[1]    (row of 10 integers)
```

Source: foo (a scalar type: int)
 Destination: bar (a scalar type: char)

Source: two_d_array[1][2] (single element of type int)
 Destination: bar (single integer)

The following modification is *not* permitted:

Source: two_d_array[1] (row of 10 integers)
 Destination: foo (single integer)

4.12 Display formats for source code symbols

When displaying an object, idebug will, where possible, also display type information for an object (e.g. unsigned char).

4.12.1 Notation

The debugger uses the following symbols in its display of values:

{ }	indicates a list of values, or a structure
{ }...	indicates a character list of unknown length terminated by a null character (which is shown)
'c'	indicates a character
'\HH'	indicates a hexadecimal character
" "	indicates a character string in an array of known size
" "...	indicates a character string of unknown length terminated by a null character (which is not shown)
< >	indicates the contents of a basic or channel object which is addressed by a pointer (except when the object is volatile)
()	provides extra information about an object

In addition, in the descriptions below, the following notation is used:

ddd	indicates a (possibly signed) decimal value
0xHHH	indicates a hexadecimal value
fff	indicates a floating point number of the form: ddd.ddd or ddd.dddEddd

4.12.2 Basic Types

Display formats for basic C types are given in table 4.9. Displays are given in normal decimal format and in hex format (invoked by `TOGGLE HEX`).

Type	Hex print off	Hex print on
<code>char</code>	<i>ddd ('c')</i> type	<i>ddd ('\xHH')</i> type
<code>short</code> <code>int</code> <code>long</code>	<i>ddd</i> type	<i>0xHHH (ddd)</i> type
<code>float</code>	<i>fff</i> float	<i>fff (0xHHH)</i> float
<code>double</code> <code>long double</code>	<i>fff</i> double	<i>fff (0xHHH)</i> double
For <code>char</code> , <i>type</i> is either <code>signed char</code> , or <code>unsigned char</code> . For integral types, <i>type</i> is either just the type name, or <code>unsigned</code> followed by the type name.		

Table 4.9 Basic C types

4.12.3 Default type of “plain” char

By default, the type of a `char` (known as a “plain” `char`) typed into `idebug` is `unsigned char`. This matches the default implemented by the INMOS C compiler `icc`. If however, the default type of plain chars has been overridden with the C compiler `'fC'` option, it may be necessary to override the default type in `idebug` by use of a cast. For example:

```
(signed char) 'c'
```

Note that such a cast is only necessary for a plain `char` entered by hand: `idebug` will correctly interpret the type of a plain `char` identifier contained in program code.

4.12.4 Enumerated types

Variables of an enumerated type are displayed as their integer value (in exactly the same manner as an `int`) followed by the name of the enumeration and the enumeration constant name for the value. If there are multiple enumerated constants that share the same value, a list is formed containing all of the enumeration constant names. `invalid enum constant` is used to indicate when a value is not a member of an enumerated type.

```
integer (enum-tag-name: enum-const-name)
```

```
integer (enum-tag-name: {enum-const-name, ...})
```

```
integer (enum-tag-name: invalid enum constant)
```

4.12.5 Pointers

Pointers are displayed in one of the following ways:

```

0xHHH (null pointer)
0xHHH (pointer to const volatile)
0xHHH (pointer to volatile)
0xHHH (channel pointer to link)
0xHHH (channel pointer to idebug virtual link link)
0xHHH (channel pointer to software virtual link link)
0xHHH (channel pointer to Event in)
0xHHH (channel pointer)
0xHHH * (mis-aligned pointer)
0xHHH < content of basic object >
0xHHH (pointer)

```

4.12.6 Function Pointers

If the function pointed to is defined in the current module, then the name of the function is displayed.

```

0xHHH (function pointer to "functionname ()")
0xHHH (cannot find corresponding function)

```

4.12.7 Structs

Members of structures are described as for other identifiers of the appropriate type. In order to display structures in a readable manner, members which are derived types are not always displayed in as much detail as when the member occurs on its own. To obtain more detail, inspect the member of the structure explicitly.

```

identifier = {
    member1
    member2
    member3
    :
}

```

For large structures, idebug pages the display and requests confirmation to continue after each page.

4.12.8 Unions

Unions are displayed in the same manner as structs except that a question mark ? appears to the left of each member to indicate that only one member of the union should be selected.

```

identifier = {
    ?    member1
    ?    member2
    ?    member3
    ?    :
}

```

4.12.9 Addressof operator &

The result of the *addressof* operator, *&*, is automatically printed as both a hex and integer value regardless of the setting of `TOGGLE HEX`. Note that the result type of *addressof* is an `int` rather than a pointer to the type used and is displayed in a similar way:

```
&identifier = 0xHHH (ddd)
```

4.12.10 Arrays

When displaying arrays, *idebug* prints the valid range of each dimension (if known) in addition to any type information and the contents of the array. For single dimension arrays containing *arithmetic* types each member of the array is displayed regardless of the size of the array. For large arrays *idebug* pages the display and requests confirmation to continue after each page. For large arrays, where the full display may be unwieldy, use array subranging to display the area of interest.

Single dimensional arrays of arithmetic types are displayed as:

```
identifier = array [0..M] of type {list of values}
```

Single dimensional arrays of other types are displayed as:

```
identifier = array [0..M] of type
```

Multi-dimensional arrays of all types are displayed as:

```
identifier = array [0..M][0..N] of type
```

Sub ranges of arrays are shown as follows:

```
identifier[ddd;ddd] = subarray of type {list of values}
```


4.12.11 Channels

Channels are displayed with information about the process that is waiting on the channel (or **Empty** if no is process waiting), in one of the following forms:

```
identifier = Channel <Iptr: address, Wdesc: address (Lo)>
```

```
identifier = Channel <Iptr: address, Wdesc: address (Hi)>
```

```
identifier = Channel <Empty>
```

```
identifier = Channel <Empty (Link link)>
```

```
identifier = Channel <Empty (Software virtual link link)>
```

An asterisk ***** is used to denote an incorrect **Iptr** or **Wdesc** which is not in the defined memory map range of the program but is in the defined memory range of the processor.

A double asterisk ****** is used to denote an incorrect **Iptr** or **Wdesc** which is not in the defined memory map range of the program and not in the defined memory range of the processor.

```
channel (name) = Channel <Iptr: addr*, Wdesc: addr**>
```

4.13 Example displays

Tables 4.10 and 4.11 show the display formats for a number of types, using the following source code segment compiled for a 32 bit transputer (for a 16 bit transputer, addresses and integers in hex format would be displayed with 4 hex digits instead of 8). The program containing this code is provided as `display.c` in the C toolset debugger examples directory.

```
enum colour { red = 1 };
struct Many {
    int    a;
    double b;
};

enum colour    shoe = red;
struct Many    many = { -42, 2.0 };

int    answer    = 42;
char    key      = 'A';
char    string[] = "bye";
char*    ptr      = string;
short    iarray[] = { 1, 2, 3, 4 };
```

Expression	Display (hex print off)
<code>answer</code>	<code>answer = 42 int</code>
<code>&answer</code>	<code>&answer = 0x801FFF2C (-2145386708) int</code>
<code>key</code>	<code>key 65 ('A') unsigned char</code>
<code>string</code>	<code>string = array [0..3] of unsigned char "bye\0"</code>
<code>ptr</code>	<code>ptr = 0x801FFF18 < "bye"... unsigned char ></code>
<code>array[1;2]</code>	<code>array[1;2] = subarray of short {2, 3}</code>
<code>shoe</code>	<code>shoe = 1 int (colour: red)</code>
<code>red</code>	<code>red = 1 int (enum constant)</code>
<code>many</code>	<pre>many = { a = -42 int b = 2.0 double }</pre>

Table 4.10 Display formats with hex printing off

Expression	Display (hex print on)
<code>answer</code>	<code>answer = 0x0000002A (42) int</code>
<code>&answer</code>	<code>&answer = 0x801FFF2C (-2145386708) int</code>
<code>key</code>	<code>key 65 ('\x41') unsigned char</code>
<code>string</code>	<code>string = array [0..3] of unsigned char {\x62, \x79, \x65, \x00}</code>
<code>ptr</code>	<code>ptr = 0x801FFF18 < {\x62, \x79, \x65, \x00}... unsigned char ></code>
<code>array[1;2]</code>	<code>array[1;2] = subarray of short {0x0002, 0x0003}</code>
<code>shoe</code>	<code>shoe = 0x00000001 (1) int (colour: red)</code>
<code>red</code>	<code>red = 0x00000001 (1) int (enum constant)</code>
<code>many</code>	<pre>many = { a = 0xFFFFFFF6 (-42) int b = 2.0 (0x4000000000000000) double }</pre>

Table 4.11 Display formats with hex printing on

4.14 INSPECT/MODIFY expression language for OCCAM

The expression language for debugging OCCAM programs is simpler than that provided for C and FORTRAN — only a single identifier name can be entered for inspection and only literal constants can be used as modification values.

4.14.1 Inspecting memory

To inspect the contents of any location in memory, specify an address rather than a symbol name. Type the address as a decimal number, a hexadecimal number (preceded by '#'), or the special short form `%h...h`, which assumes the prefix `#8000...`

The debugger displays the contents of the word of memory at that address, in both decimal and hexadecimal. For more versatile displays of memory contents, use the Monitor page commands.

4.14.2 Inspecting arrays

If the symbol inspected is an array, elements from the array can be selected using constant integer subscripts enclosed in square brackets ('[' and ']'); if no subscripts are specified the debugger prompts for them.

For short byte arrays the debugger displays the contents of the array as a string. Otherwise the debugger displays the size and type of the array, and prompts for subscript values. For example:

```
[5][4]INT ARRAY 'a', Subscripts ?
```

Press RETURN to obtain the address of the array, or enter the required subscripts, which must be in the correct range. The subscripts should be typed either as decimal constant integer values, or as integers separated by commas, for example '[3][2]', or '3, 2'. Spaces are ignored.

To simplify access to values indexed by variables (such as `'a[i]'`) an array may be indexed with '!' (e.g. `'a[!]'`) — the '!' character is replaced by the value of the last integer displayed.

Scrolling arrays

As well as simply displaying a single element of an array, the debugger allows an array to be scrolled through one element at a time. In addition, byte arrays can be displayed as a 16 byte 'segment' of the array — this segment can be moved up and down like a window into the array contents.

Array scrolling is enabled by adding '++' after the array name when prompted for a symbol name, or after the subscript when responding to the 'Subscripts ?' prompt (entering '++' alone in response to the subscript prompt assumes a subscript of zero). The debugger then displays the first array element and then the following prompt on the second line of the screen:

```
Press [UP] or [DOWN] to scroll, any other key to exit :
```

The ▲ and ▼ cursor keys can then be used to scroll through the elements of the array. The debugger will not allow you to scroll past the beginning or end of the array. Pressing any other key will return you to normal symbolic mode.

Byte arrays can be viewed in sixteen character segments by appending a '+' after the array name or subscript. As before, the cursor keys can be used to move the 'window' up and down the byte array one character at a time. Using '+' on anything other than a byte array behaves identically to using '++'.

4.14.3 Type compatibility when using MODIFY

Once a variable is selected the debugger prompts for a new value. The new value should be specified in the expected OCCam type (as specified within the prompt) although there are a few relaxations to this rule to allow for implicit casts when using the debugger (see below). **REAL32** and **REAL64** values must be given in the correct OCCam format — including a sign for the exponent, if present.

The following OCCam types may be freely mixed to provide implicit type casts so long as the value is defined within the destination type:

BOOL **BYTE** **INT** **INT16** **INT32** **INT64**

The following are examples of valid modification values:

Type	Modify value
REAL32	42.0
INT64	TRUE
INT	'a'
BOOL	1
BOOL	'*#00'
INT16	#A0
INT32	\$1A
BYTE	42

The following are examples of *invalid* modification values:

Type	Modify value
REAL32	42
INT64	2.0
BOOL	'*#02'
INT16	32768
BYTE	-1
BYTE	#100

4.15 Display formats for source code symbols

When displaying an object, `idebug` will display its type and value, together with its address in memory. If there is too much information to be displayed on one line, it is displayed in two parts. The symbol's name and type is displayed first and then, after a short pause, the value and address.

The display formats for the basic OCCAM types (`BOOL`, `BYTE`, `INT`, `INT16`, `INT32`, `INT64`, `REAL32` and `REAL64`), channels, arrays, and procedures and functions are described below. For protocol names and tags, and timers only the type and name are displayed.

4.15.1 Notation

The debugger uses the following symbols in its display of values:

<code>' c'</code>	indicates a character
<code>" "</code>	indicates a character string
<code>()</code>	provides extra information about an object
<code>(at #hhh)</code>	shows the memory address of an object

In addition, in the descriptions below, the following notation is used:

<code>ddd</code>	indicates a (possibly signed) decimal value
<code>#hhh</code>	indicates a hexadecimal value
<code>fff</code>	indicates a floating point number of the form: <code>ddd.ddd</code> or <code>ddd.dddEddd</code>
<code>bbb</code>	indicates a boolean value (<code>TRUE</code> or <code>FALSE</code>)

4.15.2 Basic Types

Display formats for the basic OCCAM types are given in table 4.12. When debugging OCCAM programs, the debugger always displays both decimal and hexadecimal values for integer types (regardless of the state invoked by `TOGGLE HEX`).

Type	Display
BYTE	BYTE ' <i>name</i> ' has value <i>ddd</i> (<i>#hh</i> , ' <i>c</i> ') (at <i>#hhh</i>) Note: non-printing characters are displayed as '.'
BOOL	BOOL ' <i>name</i> ' has value <i>bbb</i> (at <i>#hhh</i>)
INT INT16 INT32 INT64	type ' <i>name</i> ' has value <i>ddd</i> (<i>#hhh</i>) (at <i>#hhh</i>)
REAL32 REAL64	type ' <i>name</i> ' has value <i>fff</i> (<i>#hhh</i>) (at <i>#hhh</i>)

Table 4.12 Display formats for basic occam types

If a variable is optimized out because it is not used in the program, then the following message is displayed:

type 'name' was never used and has been optimised out.

4.15.3 Channels

For channels, which are not empty, the *Iptr* and *Wdesc* of the process waiting for communication, and its priority, are displayed.

Channels are displayed in one of the following forms:

CHAN '*chan*' is empty (at *#hhh*)

CHAN '*chan*' has *Iptr*: *#hhh* and *Wdesc*: *#hhh* (Lo) (at *#hhh*)

CHAN '*chan*' has *Iptr*: *#hhh* and *Wdesc*: *#hhh* (Hi) (at *#hhh*)

An asterisk *** is used to denote an incorrect *Iptr* or *Wdesc* which is not in the defined memory map range of the program but is in the defined memory range of the processor.

A double asterisk **** is used to denote an incorrect *Iptr* or *Wdesc* which is not in the defined memory map range of the program and not in the defined memory range of the processor.

CHAN '*chan*' has *Iptr*: *#hhh** and *Wdesc*: *#hhh* (Hi) (at *#hhh***)

If the channel is a hard channel then information about the link (or event channel) that it is mapped onto is also provided. For example:

CHAN '*fs*' is empty (Link 1 in)

If the channel is a software virtual link provided by the configurer, then the virtual link number is displayed. However, this does not show whether this is an input or output virtual link. For example:

CHAN '*fs*' is empty (virtual Link 41 at *#hhh*)

4.15.4 Arrays

If subscripts are specified then the type, value, and address of the array element are displayed as described above.

If no subscripts are given then, for a short **BYTE** array, the contents are displayed in ASCII. For any other type of array, just the dimensions, type and address of the array are displayed.

4.15.5 Procedures and functions

For procedure or function names, the entry address, and nested workspace and vectorspace requirements are displayed (no address is displayed for library names):

```
PROC 'name' at #hhh, uses ddd WS and ddd VS slots
```

```
FUNCTION 'name' at #hhh, uses ddd WS and ddd VS slots
```

4.16 Example displays

Table 4.13 shows the display formats for a number of types, using the following source code segment compiled for a 32 bit transputer (for a 16 bit transputer, addresses and integers in hex format would be displayed with 4 hex digits instead of 8).

```

-----
--
--  Debugger example:  display.occ
--
--  Example of occam display types within idebug.
--
--  Note: This example uses the PAR construct in an
--        inefficient manner for illustrative purposes only.
--
-----

#include "hostio.inc"
#include "linkaddr.inc"
#USE     "hostio.lib"
#USE     "debug.lib"

PROC example (CHAN OF SP fs, ts, []INT free.memory)
  VAL name IS "occam example" :

  CHAN OF INT ca, cb :
  PLACE ca AT event.in :
  BOOL bool :
  BYTE byte :
  INT int :
  INT16 int16 :
  INT32 int32 :
  INT64 int64 :
  REAL32 real32 :
  REAL64 real64 :
  [20][32]INT grid :
  [256]BYTE string :
  INT x, y :
  INT not.used :
  SEQ
    PAR
      ca ? x
      cb ? y
      bool := TRUE
      byte := 'B'
      int := -42
      int16 := -42 (INT16)
      int32 := -42 (INT32)
      int64 := -42 (INT64)
      real32 := 1.0 (REAL32)
      real64 := 1.0 (REAL64)
      grid[0] := grid[1]
      [string FROM 0 FOR SIZE name] := name
    IF
      (SIZE free.memory) > 0
        free.memory[0] := 66
      TRUE
      SKIP

```



```

        DEBUG.STOP ()    --  debugger will locate to here
        so.exit(fs, ts, sps.success)
:

```

Symbol	Display
ca	CHAN 'ca' ... (then, after a pause) has Iptr: #80003EAE and Wdesc: #80003DE5 (Lo) (PLACED AT 8) (Event in)
cb	CHAN 'cb' has Iptr: #80003EB5 and Wdesc: #80003DD9 (Lo) (at #80003E24)
not.used	INT 'not.used' was never used and has been optimised out
bool	BOOL 'bool' has value TRUE (at #80003E20)
byte	BYTE 'byte' has value 66 (#42, 'B') (at #80003E1C)
int	INT 'int' has value -42 (FFFFFFD6) (at #80003E18)
int16	INT16 'int16' has value -42 (FFD6) (at #80003E14)
int32	INT32 'int32' has value -42 (FFFFFFD6) (at #80003E10)
int64	INT64 'int64' has value -42 (FFFFFFFFFFFFFFD6) (at #80003E08)
real32	REAL32 'real32' has value 1.0 (#3F800000) (at #80003E04)
real64	REAL64 'real64' has value 1.0 (#3FF0000000000000) (at #80003DFC)
grid	[20][32]INT ARRAY 'grid' (at #800041D8)
string	[256]BYTE ARRAY 'string' (at #80004BD8)
string +	[16]BYTES from 'string[0]' is "occam example..." (at #80004BD8) Press [UP] or [DOWN] to scroll, any other key to exit :
string ++	BYTE 'string[0]' has value 111 (#6F, 'o') (at #80004BD8) Press [UP] or [DOWN] to scroll, any other key to exit :
example	PROC 'example' at #80003E4C, uses 74 WS and 706 VS slots
DEBUG.STOP	LIB PROC 'DEBUG.STOP' uses 25 WS and 0 VS slots

Table 4.13 occam display formats

4.17 Error messages

This section lists error messages generated by `idebug`. Other messages not in this list may be generated by corrupt files and by files not created by the toolset.

4.17.1 Out of memory errors

If the debugger runs out of memory when trying to read in information and the offending code module cannot be reduced in size, the amount of memory available to the debugger may be increased by increasing the size of the memory on the transputer the debugger is running on and updating `DEBUGSIZE` accordingly.

4.17.2 If the debugger hangs

If the debugger starts up but then hangs with the message:

Loading network...

one of the following errors may have occurred:

- 1 The network connectivity is not correctly described in the configuration description, for example, a link is not connected to a processor, or the type of a processor has been specified incorrectly.

Network connectivity on a board can be checked by running a check or worm program, such as the `ispy` program supplied with the support software for some INMOS *iq* systems products. These products are available separately from your local INMOS distributor.

- 2 You have set `DEBUGSIZE` to be larger than the memory on the root processor (where the debugger is running).

Change `DEBUGSIZE` to reflect the correct root processor memory size.

4.17.3 Error message list

"filename" not compiled with full symbolic debug information

The object code module does not contain sufficient debug information for the debugger to locate to its corresponding source code (i.e. it contains minimal debug information). Recompile the module and rebuild the program in order to debug it symbolically.

Already located – No process is waiting at the other end of this link

An attempt to jump down a hard channel (link) has failed because there is no process waiting at the other end.

Attempted read outside Parameter block
Attempted write outside Parameter block

The configuration system has become corrupted. Check hardware using a memory check program such as *ispy*. (The *ispy* program is supplied as part of the board support software for INMOS *iq* systems products. These products are available separately from your local INMOS distributor.)

Can only specify a transputer type if bootable is for a class

Cannot specify a transputer type for configured bootable files

You have tried to specify a processor type when the bootable file is already for a specific processor type.

Cannot create network dump – *reason*

Creation of a network dump file is not permitted on a program that is, or has been, interactively debugged. *reason* can be either of the following:

- 1 when in Interactive mode
- 2 when in Postmortem Interactive mode
- 3 Already reading one

Cannot debug boot from ROM run in ROM file "*filename*"

You may only debug boot from ROM run in RAM programs with *idebug*.

Cannot find this line's location

Either of the following has occurred:

- 1 You have moved the cursor beyond the end of the current source file for which there is no executable code.
- 2 The compiler has optimized the executable code out.

Cannot locate beyond Freespace area

The address specified is not within the memory map range of the processor.

Cannot locate to area (lptr: *#address*)

The address specified is not within the code area for the program on the processor. *area* can be any of the following:

Reserved transputer memory
Runtime kernel

Reserved memory
Configuration code area
Stack area
Vectorspace area
Static area
Heap area
Freespace area

Cannot open "*filename*"

Either the file does not exist or it is not on the **ISEARCH** path (note that by default this includes the current directory). The **ilist** tool can be used to confirm this.

Note: if the file name is **vrdebxx.tco** (or something similar), where **xx** is a sequence of digits, then you are probably trying to locate to one of the configurator's software virtual link processes. Use the **[Z]** command to display processes waiting on the software virtual links.

Cannot read processor *number* (Txx)

The debugger cannot communicate with that processor. Any of the following errors may have occurred:

- 1 The root processor's core dump has been incorrectly specified.
- 2 The debugger has failed to analyze the network correctly. Either you have failed to specify the '**A**' option or the system control signals are wired incorrectly.
- 3 The network does not match that specified in the configuration file. Check network connectivity using a check program such as **ispy**. (The **ispy** program is supplied as part of the board support software for INMOS *iq* systems products. These products are available separately from your local INMOS distributor.)

Cannot run application – the program has crashed !

Use the **[Y]** (Enter post-mortem debugging) Monitor page command to post-mortem debug the (now defunct) breakpoint session.

Channel is invalid

The channel does not point to a known process executing on the processor.

Configuration info inconsistent with linked unit

You have probably relinked a component of a program and forgotten to reconfigure it.

Configured for post-mortem debugging only

You have explicitly disabled interactive debugging (using configurer or collector options).

Debug info too large (*reason*)

The debugging information for a particular compilation module is too large for the debugger. Either reduce the size of the offending module or increase the size of memory on the processor where the debugger is running (see section 4.17.1 on how to overcome this).

reason can be any of the following:

ix.tags is full
ws.array is full
name table is full

Debugger incompatible configuration file "*filename*"

You have configured your program without specifying the debugger compatible option ('G' option) to the configurer, *icconf*.

For mixed language programs configured with the occam toolset configurer *occonf*, the error may be that you have configured your program with the configurer 'RE' option to enable memory layout re-ordering.

File has changed since configuration "*filename*"

You should rebuild the program again.

FILE IS TOO BIG – truncated

The debugger buffer capacity has been exceeded. The buffer contains as much of the file as could be read before the capacity was exceeded (see section 4.17.1).

Illegal virtual channel address

The channel has been (possibly incorrectly) tagged as virtual but does not point to a valid software virtual channel (as defined by the debugging kernel or the configurer). This is caused by a channel that has become corrupted (normally by overwriting the location of the channel). You should ensure that no compiler checks have been disabled to prevent accidental corruption.

Incompatible debugger modes: *message*

Mutually incompatible options have been specified on the command line.

Interactive debugging has been disabled

The module has been linked with the linker 'Y' option to disable breakpoint (interactive) debugging. Rebuild your program without disabling interactive debugging and retry.

ITERM error on line *linenumber*, *message*

The debugger has detected a syntax error in the ITERM file. *message* describes the error.

Name *symbol* is not in dynamic scope

The symbol *symbol* exists in the module, but is not in scope from where the debugger last located to. In order to inspect the symbol you must locate to a new position where the symbol is in scope.

Not a (compatible) bootable file "*filename*"

The file is either a non-bootable file or a pre-product release bootable file. Use `ilist` to determine the contents of the file if in doubt.

Not enough free memory for the debugger

You have either not set the environment variable `IDEBUGSIZE` or you have set it to be too small (it should be > 400K). Change the variable to reflect the memory size of the root processor.

Not on a valid INCLUDE line

You may only use `ENTER FILE` when the cursor is on a line with an include directive.

Only debugging tools and cursor keys are available

You have pressed a key which is not defined.

Option must be followed by a link number (0 – 3)

Command line options 'B', 'M', and 'T' require a link number in the range 0 to 3.

Option must be followed by a valid processor type (eg. T425)

The processor type supplied is not recognized by the debugger.

(Probe Go) : Processor *number* – Cannot contact

The debugger is unable to communicate with processor *number*. The processor type specified in the configuration (or to the debugger via the 'c' option) does not match that found. Check the network using a program such as `ispy` in order to determine the correct processor type.

(Probe Go) : Processor *number* – Expected processor type Txxx, found Txxx

The processor type specified in the configuration (or to the debugger via the 'c' option) does not match that found. Check the configuration descrip-

tion and the network (using a program such as `ispy`) in order to determine the correct processor type.

(Probe Resume) : Processor *number* – Invalid Breakpoint

The debugger has stopped at a breakpoint which it did not place in the code. If you wish to continue executing the program set a breakpoint at the same address and retry the command.

Processor *number* : insufficient memory, require at least *number* bytes

The memory requirement of the processor as specified to the configurer, the collector, or in `IBOARDSIZE` (as appropriate) is too small. (Note that the value displayed may include memory for some configuration code that is reclaimed when program starts executing.)

This may also be caused by the debugging Runtime kernel using an extra 11—15K of memory.

Processor type must be a 32 bit processor (eg. T425)

You must specify a 32 bit processor type because processor classes are for 32 bit processors only.

Processor type must be not abbreviated

You must specify specific processor types rather than abbreviated types (e.g. T425 rather than T5) because some abbreviated types cover more than one specific type.

READ ERROR – truncated

The debugger could not read all of the file. The buffer contains as much of the file as could be read (see section 4.17.1 on how to overcome this).

Runtime kernel is not present (or has been overwritten)

Either the runtime kernel has been corrupted or you are trying to postmortem a breakpoint session that didn't occur.

There is no enclosing INCLUDE

You have attempted to use `EXIT FILE` when not located in a nested include file.

There are no processes waiting at either end of this link

An attempt to jump down a hard channel (link) has failed because there are no processes waiting at either end.

This transputer link is connected to the host

The link specified in the 'B', 'M', or 'T' command line option is the communication link from the debugger to the host and is not connected to the network.

Too many processes declared at configuration level (*number*)
Too many processes used at configuration level (*number*)

The debugger requires more memory in order to operate on this many processes (see section 4.17.1 on how to overcome this).

Too many processors – There is only enough room for (*number*)

The debugger requires more memory in order to operate on this many processors (see section 4.17.1 on how to overcome this).

Unable to find any low priority entryptoints on any processor
Unable to find any low priority entryptoints on this processor
Unable to find any low priority main () on this processor

The processes you have requested the debugger to set breakpoints in are all at high priority on processors with no hardware breakpoint support.

Unable to read environment variable ITERM

There is no translation for the ITERM environment variable which defines the screen and keyboard format.

Unable to toggle a breakpoint on this line

The breakpoint cannot be set or cleared on this source line. Either:

- 1 The current file contains no executable code, or
- 2 Executable code is contained in an include file and the debugger cannot determine whether you mean to toggle the breakpoint in that file or in the current file.

Move to the line where you really want to toggle the breakpoint and retry the command.

Unknown core dump format "*filename*"

The network dump file is in the wrong format or the wrong file has been specified. You can use `ilist` to determine the format of the file.

Wdesc is invalid – *message*

The `wdesc` supplied is invalid: this may be deliberate because it is unknown. If you supplied it from the Monitor page environment, retry the command with a valid `wdesc`.

message can be one of:

cannot inspect variables
cannot modify variables

cannot backtrace
cannot auto backtrace out of library

Wrong number of processors in network dump file filename

The number of processors does not correspond to the current program.
The wrong network dump file may have been specified.

You cannot backtrace from here (to configuration code)

This normally occurs when you try to backtrace from the program's topmost procedure into the bootstrap routine which is not supported symbolically by the debugger (i.e. the configuration code area).

You cannot backtrace from here (to `lptr: #nnn, Wdesc: #mmm`)

An attempt to backtrace from a procedure or function has failed because the resultant process details are invalid (e.g. `lptr` is not in the Code area), or you are trying to locate to a software virtual link router process..

The `lptr` and `Wdesc` shown are those of the invalid process which supposedly called the current procedure or function.

If this error occurs, you should use `INFO` before backtracing to check that the current process details are valid (they are normally only invalid when incorrect process details have been specified with the Monitor page 'O' command). Corruption of the stack (workspace) is another possible cause of this error; to prevent accidental corruption you should ensure that no compiler checks have been disabled.

This error can also occur if you try to locate to a process which implements the software virtual links. This can be checked by using the `V` command to search for a process with a Code area which contains the displayed `lptr` and a stack area which contains the displayed `Wdesc`. If the name of the process is "`%ROUTER[n]`" then it is a software virtual link process which can not be located to.

You have changed file, so you can't use this key

There are certain symbolic features that cannot be used if you have changed to another source file. Either use `RELOCATE`, or relocate to the original file using the Monitor page `F` (Select file) command, before retrying the command.

You must specify a filename

The command line syntax requires a filename.

You must specify a transputer type (bootable is for a TA class)

You must specify a transputer type (bootable is for a TB class)

The program you are trying to debug is for a transputer class (either TA or TB); the debugger needs to know the actual processor type (e.g. T425).

You should retry using the debugger with the command line 'C' option to specify the appropriate processor type.

You must specify the application boardsize in IBOARDSize to be <= #10000

On a T2 the maximum memory size is 64K (#10000).

5 `idump` — memory dumper

This chapter describes the memory dumper tool `idump` that dumps the contents of the root processor's memory to disk. It is used to enable the debugging of code running on the root transputer.

5.1 Introduction

The memory dumper allows programs that use the root transputer to be debugged in the normal way using the debugger tool `idebug`. It is required because `idebug` runs on the root transputer and overwrites all code and code in its memory. `idump` saves the contents of the root transputer to a disk file in a format that can be read by the debugger. Information contained in the file allows the debugger to analyze data in the root transputer in the same manner as other transputers on the network.

When `idump` is invoked it calls the server with the '`SA`' option so that the space occupied by the dumper program is saved before it is loaded onto the transputer.

5.2 Running the memory dumper

To invoke the `idump` tool, use the following command line:

```
► idump filename memorysize [{ startoffset length }]
```

where: *filename* is the name of the dump file to be created.

memorysize is the number of bytes, starting at the bottom of memory, to be written to the file.

startoffset is an offset in bytes from the start of memory.

length is the amount of memory in bytes, starting at *startoffset*, to be dumped in addition to *memorysize*.

All parameters can be expressed in either decimal or in hexadecimal format. Hexadecimal numbers must be preceded by the hash character '`#`' or the dollar sign '`$`'.

The memory dump file stores the contents of the transputer's registers and the first *memorysize* bytes of memory. The file is given the `.dump` extension. After the dump has been performed `idump` remains resident on the transputer board ready to load the debugger.

memorysize must be large enough to contain the complete program with its work-space and vectorspace. If the program to be dumped uses the free memory buffer, the whole of the transputer board's memory should be dumped.

Further portions of memory can be dumped by specifying the start of the segment of memory to be dumped and the number of bytes, using pairs of *startoffset* *length* parameters. The start address is given by *startoffset* and the number of bytes by *length*. The overall size of the memory dump file is given by the amount of memory saved plus around 500 bytes for the register contents.

5.2.1 Example of use

Assuming a value of 100K for `IBOARDSIZE`:

```
idump core 102400
```

This command writes the contents of the root transputer's memory to the file `core.dmp`. The `.dmp` extension is added by default because the filename is specified with no extension.

5.3 Error messages

Badly formed command line

Command line error. The command syntax requires a file name followed by the number of bytes of memory to dump. Check the syntax of the command and retry.

Cannot open file

File system error. The memory dump file could not be opened on the host system.

Cannot write file

File system error. The memory dump file could not be written to the host system.

You must tell the server to peek the transputer

`idump` has been invoked by calling the host file server with the incorrect option. This error can only occur if the tool is not invoked with the supplied executable file `idump.exe`.

6 **iemit** — memory interface configurer

This chapter describes the memory configuration tool **iemit**. This tool can be used interactively to explore the effects of changes in the external memory interface parameters of certain 32 bit transputers. The tool can also be used in batch mode to create ASCII or PostScript files. The tool produces a memory configuration file which may be included as an input file to **ieprom** and blown into EPROM along with a ROM-bootable application file.

The chapter describes how to use **iemit** and outlines its capabilities. Example displays are provided, followed by a list of error messages which the tool may generate. The format of the memory configuration file is described and an example is given. **Note:** memory configuration files are simple text files which may be created manually using a standard text editor or generated by using **iemit**.

6.1 Introduction

The IMS T400, T414, T425, T426, T800 and T805 transputers have a configurable external memory interface (EMI) which allows a variety of types of memory device to be connected using few extra components.

For these transputers, the interface configuration may be selected by one of two mechanisms. The user may select one of the 17 standard memory configurations (13 for the T414) or a customized memory configuration may be loaded from a ROM or PAL on reset.

Both methods of memory configuration are available when booting from ROM or from link. If the transputer is being booted from ROM, a customized memory configuration may be added to the ROM or a standard configuration may be used. If the transputer is booted from link a standard configuration may be used at no extra cost, or a dedicated ROM or PAL may be added for a customized configuration.

In order to generate a customized configuration the user may create a memory configuration file, describing the memory configuration and have this blown into an EPROM. The configuration chosen is made known to the transputer by simple board level connections which are detected by the transputer on reset. If a standard configuration is required the **MemConfig** pin is connected to the appropriate address pin. For example, standard configuration 7 is selected via address pin **MemAD7**. If a customized configuration is required the **MemConfig** pin is connected though an inverter to the appropriate data line, usually this is **MemnotWrD0**. **Note:** when **iemit** is used to generate the memory configuration, the **MemnotWrD0** pin must be used. For further details see *The transputer data-book*.

The external memory interface configuration tool `iemit` produces timing diagrams for potential configurations of the memory interface and warns of possible errors in the design. It indicates whether one of the preset configurations that are available would be suitable, or whether it would be necessary to use an externally programmed configuration.

Note: That it is assumed that readers creating memory configuration files are familiar with the memory interface of the processor that they are using. The stages in designing a memory interface, including examples, are described in chapter 2 of *The transputer applications notebook - Systems and performance*. Further information may also be found in *The transputer databook*.

6.2 Running `iemit`

The `iemit` tool can be invoked by the following command line:

► `iemit options`

where: *options* is a list of options given in Table 6.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

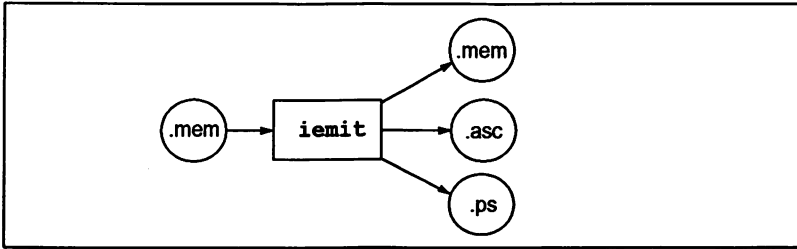
If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
A	Produce ASCII output file.
E	Invoke interactive mode.
F filename	Specify input memory configuration file.
I	Select verbose mode. In this mode the user will receive status information about what the tool is doing during operation for example, reading or writing to a file.
O filename	Specify output filename.
P	Produce PostScript output file.

Table 6.1 `iemit` command line options

Note: that if option 'E' is selected i.e. interactive mode, then *no* other options may be specified on the command line.

The operation of `iemit` in terms of standard file extensions is shown below:



Examples of use

iemit may be invoked in interactive mode by using one of the following commands:

<code>iemit -e</code>	(UNIX based toolsets)
<code>iemit /e</code>	(MS-DOS and VMS based toolsets)

Output files in ASCII or PostScript may be specified by command options from within interactive mode; alternatively **iemit** may be invoked in batch mode, to create an output file in one of these formats.

When the tool is invoked in batch mode to produce an output file in either ASCII or PostScript format, then an input file must be supplied using the '**f**' option. It is also mandatory to specify either the '**A**' or '**P**' option. If the '**O**' parameter is not supplied then an output filename will be constructed, from the input filename, with an extension of '**.ps**' for a PostScript output, or '**.asc**' for an ASCII output.

Example:

The following commands cause **iemit** to produce an output file in PostScript format. The tool is invoked in verbose mode.

UNIX based toolsets:

```
iemit -i -p -f memconfig.mem -o waveform.ps
```

MS-DOS and VMS based toolsets:

```
iemit /i /p /f memconfig.mem /o waveform.ps
```

Note: **iemit** will make use of the **ITERM** host environment variable, if it is available, otherwise it will use defaults.

6.3 Output files

Two different types of output may be produced by `iemit`, these are listed below:

- A memory configuration file suitable for including as an input file to the `ieprom` tool.
- An output file in either ASCII or Postscript format, suitable for inclusion in documentation.

The tool may be used interactively to produce a memory configuration file in text format. This file may then be used as an input file to the `ieprom` tool, thus enabling the memory configuration to be stored on ROM. `iemit` is capable of saving and reloading configurations to allow for design over an extended period and for comparison of different configurations. The memory configuration file is described and an example is given in section 6.6.

Additionally `iemit` may be used to produce an output file which is either a plain ASCII file containing timing data or a file in PostScript format containing waveform diagrams. These formats were chosen so that the results of the program could be easily included in reports or other documentation.

6.4 Interactive operation

When `iemit` is invoked in interactive mode the program will start up with the default standard configuration 31.

The tool's user interface is presented as a number of display pages showing timing data. The displays may be updated by changing the timing parameters, which are accessed from page 1. All inputs are executed immediately so that the user can see the effect on any of the displays. As each page is shown, the user has the option of selecting another page for display by keying in its number. The current configuration may be saved at any time to a specified output file.

The information displayed and options available on each page are described below.

6.4.1 Page 0

This page acts as an index to the others. It shows the title of each page and allows one of them to be selected. An option is provided to enable an input file to initialize the memory configuration. Other options enable the user to selectively generate output files. Options are listed in table 6.2 and an example of the display page is given in figure 6.1.

The user enters an option code followed by the `RETURN` key. If a file option is specified the user will be prompted for a filename. **Note:** file extensions should be specified, there are no defaults.

Option	Description
1 to 6	Selects the page to be displayed.
Q	Quit - selection of this option exits the program.
L	<p>Load previously saved configuration. A filename is prompted for, and the configuration saved in that file is read in and the display data is updated. The program expects a memory configuration file.</p> <p>If loading does not succeed because the file has a bad format, the current configuration is reset to standard configuration 31. If loading fails because the file could not be found or could not be opened for reading, the load is abandoned without losing the current configuration.</p>
S	Save configuration to a file. The program prompts for the name of a file to which the data will be written, by convention the extension .mem should be used. Output is a memory configuration file. An error is reported if the data could not be saved. The saved file is given comments in its header indicating that it was created by the iemit program.
A	Output pages in ASCII format to a file. The program prompts for the name of a file to which the data will be written. Output is in plain ASCII format with a form feed (FF) character after each page. It includes full timing information and a representation of the timing diagrams for read and write cycles. An error is reported if the output could not be written.
P	<p>Generate PostScript file. The program prompts for a filename. The program writes to the file a program in the PostScript page description language to draw the timing diagrams for the chosen memory interface configuration. The waveforms shown are the same as those which can be seen by selecting pages 4 and 5.</p> <p>The file produced fully conforms to the PostScript structuring conventions, allowing it to be processed by other programs. The diagram is designed to fit lengthways on an A4 page, and is suitable for inclusion in technical notes and reports. The file can be sent directly to an Apple LaserWriter or other PostScript output device.</p>

Table 6.2 iemit page 0 options

```

Page 0

T414/T800 External Memory Interface Program
=====

Page 0:  Index - this page
        1:  EMI configuration parameters
        2:  General timing
        3:  Dynamic RAM timing
        4:  Read cycle waveforms
        5:  Write cycle waveforms
        6:  Configuration table

Please enter 1...6 to see a new page;
      <S> to save configuration to a file;
      <L> to load a saved configuration;
      <A> to generate an ASCII listing of all pages to a file;
      <P> to generate PostScript file of waveforms;
      <Q> to exit the program.
:

```

Figure 6.1 Example iemit display page 0

6.4.2 Page 1

This page shows the input parameters to `iemit`. It is from these parameters that the tool computes the timing information and the waveforms. Only one parameter may be changed at a time and the display data is immediately updated. An example of the display page is given in figure 6.2.

```

Page 1

EMI configuration parameters
=====
Device type                T425-25
EMI clock period Tm        20ns at ClockIn = 5MHz
Wait states                0
Address setup time         T1: 4 periods Tm
Address hold time          T2: 4 periods Tm
Read cycle tristate/write data setup T3: 4 periods Tm
Extended for wait          T4: 4 periods Tm
Read or write data         T5: 4 periods Tm
End tristate/data hold     T6: 4 periods Tm
Nonprogrammable strobe "notMemS0" "0" S0
Programmable strobe "notMemS1" "1" S1: 30 periods Tm
Programmable strobe "notMemS2" "2" S2: 30 periods Tm
Programmable strobe "notMemS3" "3" S3: 30 periods Tm
Programmable strobe "notMemS4" "4" S4: 18 periods Tm
Read cycle strobe "notMemRd" "r"
Write cycle strobe "notMemWrB" "w"
Refresh period: 72 ClockIn periods      Wait: 0
Write mode: Late                        Configuration: 31

Enter a new page number (0 for the index) or <C> to change a parameter:

```

Figure 6.2 Example iemit display page 1

When the page is displayed, the user has the option to select a new page by entering its number, or entering **C** to change one of the parameters. In the latter case,

a list of parameter identifiers is displayed (see table 6.3) and the user is prompted to select one. The user may then specify a new value, or by pressing the **RETURN** key, leave the current selection unchanged. The parameters used for modifying the timing data are described in tables 6.4, and 6.5.

Parameter identifier	Parameter
0 to 6	Page to be displayed
D	Device type
T1	Address setup time before address valid strobe
T2	Address hold time after address valid strobe
T3	Read cycle tristate or write data setup
T4	Extendible data setup time
T5	Read or write data
T6	End tristate or data hold
S0	Non-programmable strobe "notMemS0"
S1	Programmable strobe "notMemS1"
S2	Programmable strobe "notMemS2"
S3	Programmable strobe "notMemS3"
S4	Programmable strobe "notMemS4"
RS	Read cycle strobe name
WS	Write cycle strobe name
R	Refresh period
WM	Write mode
W	Memwait input connection
C	Standard configuration

Table 6.3 **iemit** page 1 parameter identifiers

Note: there are two parameters displayed on page 1 which are calculated by **iemit** and cannot be directly updated by the user; they are the EMI clock period T_m and the Wait states (see Table 6.5).

Parameter	Description																
Device type	<p>This parameter enables the program to deduce the time taken for a half cycle of the signal ProcClockOut: this is Tm, the basic unit of time of the memory interface. A menu of the available devices is displayed and the user is invited to select one:</p> <table> <tr> <td>T400-20</td><td>T800-17</td></tr> <tr> <td>T414-15</td><td>T800-20</td></tr> <tr> <td>T414-17</td><td>T800-22</td></tr> <tr> <td>T414-20</td><td>T800-25</td></tr> <tr> <td>T425-17</td><td>T800-30</td></tr> <tr> <td>T425-20</td><td>T800-35</td></tr> <tr> <td>T425-25</td><td>T805-25</td></tr> <tr> <td>T425-30</td><td>T805-30</td></tr> </table>	T400-20	T800-17	T414-15	T800-20	T414-17	T800-22	T414-20	T800-25	T425-17	T800-30	T425-20	T800-35	T425-25	T805-25	T425-30	T805-30
T400-20	T800-17																
T414-15	T800-20																
T414-17	T800-22																
T414-20	T800-25																
T425-17	T800-30																
T425-20	T800-35																
T425-25	T805-25																
T425-30	T805-30																
Tstates T1-T6	The length of each Tstate T1 to T6, is entered as a number of Tm periods between 1 and 4. (2 Tm periods = 1 clock cycle).																
Programmable Strokes S0-S4	<p>The programmed durations of the strobes notMemS0 to notMemS4. The strobes each have two names which can be altered. One which can be up to 9 characters in length, and one consisting of just one character. There should be no embedded spaces in the long names. The short names are used in the timing information on pages 2 and 3, while the long names are used to label the waveforms on pages 4 and 5, and in the PostScript output. The signal names are initialized to sensible defaults.</p> <p>Note: that S0 is a fixed strobe, so its duration cannot be changed. The duration of a strobe can be 0 to 31 Tm periods. If the value for S1 is set to zero, then notMemS1 stays high throughout the cycle; if the value for S2, S3 or S4 is set to zero, then the strobe is low for the duration of the cycle.</p>																
Read strobe name	The names for the read strobe notMemRd can be altered.																
Write strobe name	The names for the write strobe notMemWrB can be altered. Note that because the four byte write strobes have the same timing, only one is considered.																
Refresh period	The refresh period is given as a number of ClockIn periods (18, 36, 54, or 72) or as Refresh Off if zero is selected.																

Table 6.4 iemit page 1 parameters

Parameter	Description
Write mode	The write mode can be set to Early or Late to suit the type of memory being used.
Wait connection	<p>The MemWait input may be connected to one of the strobes S2, S3, S4 by entering 'S2', 'S3' or 'S4' respectively. Alternatively, by specifying a number in the range 1 to 60 MemWait may be connected to a simulated external wait state generator. This causes MemWait to be held high then to become inactive (low) a set number of Tm periods after the start of T2. Note: that this mode is not supported directly by the T414; in a final design, a circuit would have to be built to perform this function.</p> <p>If the current connection of MemWait causes the signal to become inactive just as ProcClockOut is falling during T4, a warning is given that there is a hazard of a wait race condition. This is because MemWait is sampled on the falling edge of ProcClockOut –and if the signal is changing while being sampled, the result is undefined.</p>
EMI clock period Tm	The value of Tm for a clockIn frequency of 5MHz. This is computed from the other parameters and displayed.
Wait states	The number of wait states in the current configuration. This is computed from the other parameters and displayed.
Standard configuration	<p>The parameters can all be reset to those for one of the built in configurations. There are 13 standard configurations available for the T414, valid configuration numbers being 0 to 11 and 31. For the T400, T425, T800 and the T805 there are 17 standard configurations available, valid configuration numbers being 0 to 15 and 31. If the user selects, for a T414, one of the four configurations which are not available, a message will be displayed indicating that this configuration may not be hardwired on a T414.</p> <p>If the currently set configuration happens to correspond exactly to one of the preset configurations, the tool reports the fact.</p>

Table 6.5 iemit page 1 parameters

6.4.3 Page 2

This page shows general timing information for the interface, such as delays between various strobes and required access times of the memory devices to be used. The user should adjust these figures to allow for delays in external logic.

Table 6.6 lists the timing information displayed on this page while an example of the display is given in figure 6.3.

JEDEC symbol	Parameter description
T0L0L	Cycle time (in both nanoseconds and processor cycles)
TAVQV	Address access time
T0LQV	Access time from notMemS0
TrLQV	Access time from notMemRd
TAV0L	Address setup time
T0LAX	Address hold time
TrHQX	Read data hold time
TrHQZ	Read data turn off
T0L0H	notMemS0 pulse width low
T0H0L	notMemS0 pulse width high
TrLrH	notMemRd pulse width low
TrL0H	Effective notMemRd width
T0LwL	notMemS0 to notMemWrB delay
TDVwL	Write data setup time
TwLDX	Write data hold time 1
TwHDX	Write data hold time 2
TwLwH	Write pulse width
TwL0H	Effective notMemWrB width

Table 6.6 General timing parameters

The total cycle time is given in nanoseconds and in processor clock cycles. The only option available from this page is to select another page for display.

memory devices to be used are met. The user should adjust these figures to allow for delays in external logic. Table 6.7 lists the DRAM timing parameters.

The only option available from this page is to select another page for display. An example of the display is given in figure 6.4.

JEDEC symbol	Parameter description
T1L1H	notMemS1 pulse width
T1H1L	notMemS1 precharge time
T3L3H	notMemS3 pulse width
T3H3L	notMemS3 precharge time
T1L2L	notMemS1 to notMemS2 delay
T2L3L	notMemS2 to notMemS3 delay
T1L3L	notMemS1 to notMemS3 delay
T1LQV	Access time from notMemS1
T2LQV	Access time from notMemS2
T3LQV	Access time from notMemS3
T3L1H	notMemS1 hold (from notMemS3)
T1L3H	notMemS3 hold (from notMemS1)
TwL3H	notMemWrB to notMemS3 lead time
TwL1H	notMemWrB to notMemS1 lead time
T1LwH	notMemWrB hold (from notMemS1)
T1LDX	Write data hold from notMemS1
T3HQZ	Read data turn off
TRFSH	Time for 256 refresh cycles (in microseconds)

Table 6.7 DRAM timing parameters

6.4.5 Page 4

This page shows graphically the timing for a memory read cycle. An example of the display page is given in figure 6.5.

The Tstates and strobes are labelled, and bus activity is shown. The point where data are latched into the processor is also indicated.

At the top of the page is displayed the processor clock and the Tstates, a number indicating the Tstate, 'W' indicating a wait state, and 'E' indicating a state that is inserted to ensure that T1 starts on a rising edge of the processor clock.

Below this are displayed the waveforms of the programmable strobes and the read, write and address/data strobes. Each of these strobes is labelled with the corresponding label parameter.

The point at which the read data is latched is indicated by a '^' beneath the read cycle address/data strobe.

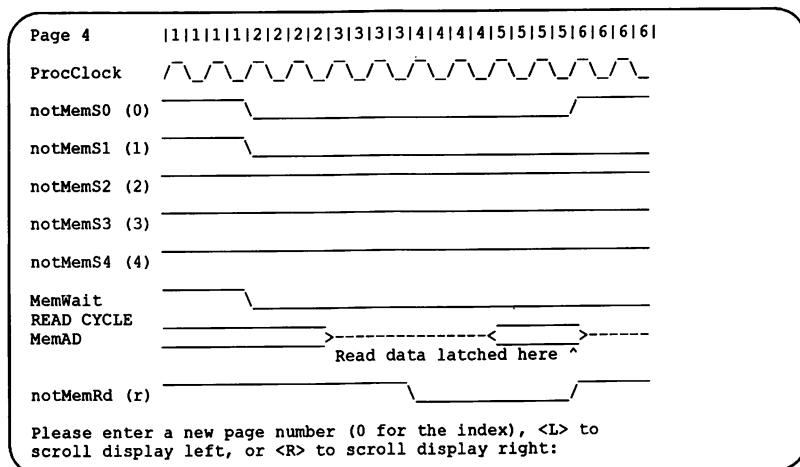


Figure 6.5 Example iemit display page 4

The MemWait waveform shows the input to the MemWait pin. If the wait input is a number then it goes low n Tm periods after the end of T1 and high again at the end of T6, if the wait input is connected to a strobe it goes low and then high when that strobe does so.

If the cycle is too long to fit horizontally on the screen, it may be scrolled left and right using the **L** and **R** keys. The displayed area moves by about 15 characters each time these are used.

6.4.6 Page 5

Page 5 shows the waveforms for a memory write cycle. The display is similar to that of page 4, indeed the read and write cycle diagrams are combined when the PostScript output is produced.

Scrolling the display to the left or right is done in the same way as for page 4.

An example of the display page is given in figure 6.6.

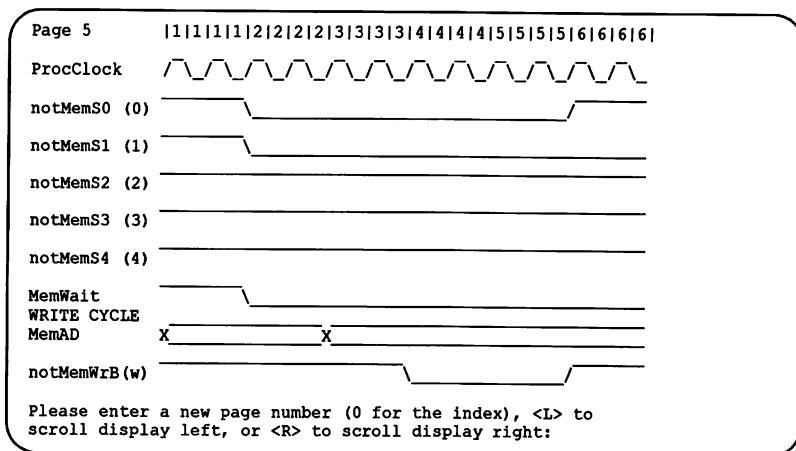


Figure 6.6 Example iemit display page 5

6.4.7 Page 6

This page gives a configuration table for the current configuration. This is a listing of the data which have to be placed in a ROM situated at the top of the transputer's memory map in order to achieve the desired configuration. The table consists of 36 words of data, but only the least significant bit in each is used. The address and contents are given for each location. **Note:** when iemit is used to generate the memory configuration, the Memconfig pin must be connected to MemnotWrD0.

An example of the display page is given in figure 6.7.

Note: that if page 1 indicates that the configuration is one of the transputer's preset ones, there will be no need for a ROM; configuration can be achieved by connecting the MemConfig pin of the device to one of the address/data lines.

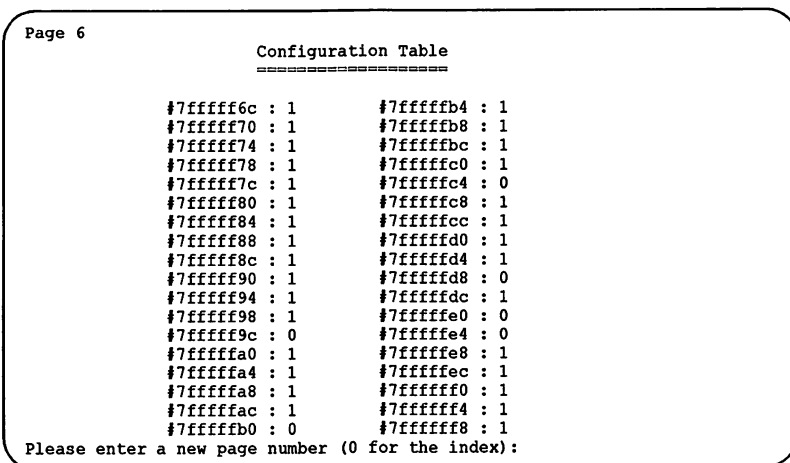


Figure 6.7 Example iemit display page 6

6.5 iemit error and warning messages

The following is a list of error and warning messages the tool can produce:

Wait race

If one of the programmable strobes is used to extend the memory cycle then the strobe must be taken low an even number of periods T_m after the start of the memory interface cycle. If the strobe is taken low an odd number of periods after the start then a wait race warning will appear. Should this warning appear, it will remain on display on page 1, until the race condition is removed. Further information can be obtained from reference 1, listed at the start of this chapter.

Input out of range

If the value entered for a numeric parameter is outside the range valid for that parameter, an input out of range warning is displayed, the value cleared from the screen and the program waits for a new value.

MemWait connection error

If an attempt is made to connect S1 to the MemWait input an error is displayed because it is a meaningless operation.

Configuration cannot be hardwired on a T414

The transputers which have a configurable memory interface all have (with the exception of the T414) 17 standard memory configurations available

to them. The T414 only has a choice of 13 standard configurations. If the standard configurations 12, 13, 14 or 15 are selected for a T414 the above warning message will be displayed against the selection on page 1.

Unable to open configuration file '*filename*'

This can occur when attempting to load a memory configuration file and indicates that the tool cannot find the specified input file. Check the spelling of the filename and/or that the file is present.

Command line parsing error

An option has been specified that the tool does not recognize.

No input file specified

This indicates that when trying to invoke the tool to produce an output file, the user has not specified a memory configuration file to use as input.

One and only one of options A or P must be specified

This indicates that when trying to produce an output file, the user has not specified whether the output is to be in ASCII or PostScript format.

Unable to open output file '*filename*'

An output filename has been specified incorrectly. Check the format of the filename.

6.6 Memory configuration file

Memory configuration files are text files which may be generated by a standard text editor or by using the memory interface configuration tool `iemit`, see section 6.2.

By convention memory configuration files have the file extension `.mem`. The file consists of a sequence of statements and comments. The following are considered to be comments:

- Blank lines
- Any line whose first significant characters are '`---`'
- Any portion of a line following '`---`'.

Comments are ignored by the `ieprom` and `iemit` tools. Statements are all other lines within the file; they may be interspersed with comments.

Individual statements are constructed of the statement and an associated parameter. These must be separated by at least one space or tab but extra spaces may

be inserted before, between, or after them for aesthetic purposes. An example memory configuration file is shown in figure 6.8.

```
-- -----
--
--      Memory configuration file produced
--      by a save command from IEMIT.
--      on Thu Feb 13 15:04:04 1992
--
-- -----

device.type      := T425-25

t1.duration      := 4
t2.duration      := 4
t3.duration      := 4
t4.duration      := 4
t5.duration      := 4
t6.duration      := 4

s0.label         := notMemS0
s1.label         := notMemS1
s2.label         := notMemS2
s3.label         := notMemS3
s4.label         := notMemS4
rs.label         := notMemRd
ws.label         := notMemWrB

s1.duration      := 30
s2.duration      := 30
s3.duration      := 30
s4.duration      := 18

refresh.period   := 72
write.mode       := LATE
wait.connection  := 0
```

Figure 6.8 Example memory configuration file

The statements defined are listed along with their parameters in table 6.8. Further information about specifying parameters is given in section 6.4.2.

Option	Description																
standard.configuration	0 to 13, or 31 for T414 processors. 0 to 15, or 31 for T400, T425, T800 and T805 processors.																
device.type	One of the following devices: <table> <tr> <td>T400-20</td><td>T800-17</td></tr> <tr> <td>T414-15</td><td>T800-20</td></tr> <tr> <td>T414-17</td><td>T800-22</td></tr> <tr> <td>T414-20</td><td>T800-25</td></tr> <tr> <td>T425-17</td><td>T800-30</td></tr> <tr> <td>T425-20</td><td>T800-35</td></tr> <tr> <td>T425-25</td><td>T800-25</td></tr> <tr> <td>T425-30</td><td>T805-30</td></tr> </table>	T400-20	T800-17	T414-15	T800-20	T414-17	T800-22	T414-20	T800-25	T425-17	T800-30	T425-20	T800-35	T425-25	T800-25	T425-30	T805-30
T400-20	T800-17																
T414-15	T800-20																
T414-17	T800-22																
T414-20	T800-25																
T425-17	T800-30																
T425-20	T800-35																
T425-25	T800-25																
T425-30	T805-30																

Option	Description
t1.duration, t2.duration, t3.duration, t4.duration, t5.duration, t6.duration	1 to 4 Tm periods. (2 Tm periods = 1 clock cycle). Defines the length in Tm periods of Tstates, T1 to T6, of the memory cycle.
s0.label, s1.label, s2.label, s3.label, s4.label	Each of these parameters accepts two text strings. They are the long (up to 9 characters) and short (1 character) names of the strobes notMemS0 to notMemS4. The names should not contain embedded spaces. Names longer than the permitted number of characters will be truncated.
rs.label	As above, the long and short names for the read strobe notMemRd.
ws.label	As above, the long and short names for the read strobe notMemWrB.
s1.duration	0 to 31 Tm periods. The S1 strobe goes low at the start of Tstate 2. This parameters defines the number of Tm periods before it goes high.
s2.duration, s3.duration, s4.duration	0 to 31 Tm periods. The S2 to S4 strobes all go high at the end of Tstate 5. These parameters define the number of Tm periods before each strobe goes low.
refresh.period	18, 36, 54, 72 or the string "Disabled". This parameter defines the period between refresh cycles as a count of ClockIn cycles.
write.mode	String value either: "Early" or "Late". Defines the write mode.
wait.connection	S2, S3, S4 or a value in the range 0 to 60. This parameter connects MemWait to one of the strobes S2, S3, S4 or to simulated external wait state generator.

Table 6.8 Memory Configuration file statements

7 `ieprom` – ROM program convertor

This chapter describes the EPROM hex tool `ieprom`. This tool is used to convert a ROM-bootable file into one or more files suitable for programming an EPROM.

The chapter describes how to invoke `ieprom` and gives details of the command line syntax. It describes the control file which the tool accepts as input and provides background information on the layout of the code in the EPROM. A description of the various file formats which may be output by the tool is given, including block mode where the output is split up over a number of files. The chapter ends with a list of error messages which may be generated by the tool.

7.1 Introduction

The INMOS EPROM software is designed so that programs which have been developed and tested using the INMOS toolset may be placed in ROM with only minor modification (see below).

This has the advantages that an application need not be committed to ROM until it is fully debugged and the actual production of the ROMs can be done relatively late in the development cycle without the fear of introducing new problems.

If a network of transputers is being used, only the root transputer needs to be booted from ROM; once this has been booted it will boot its neighbors by link.

Figure 7.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

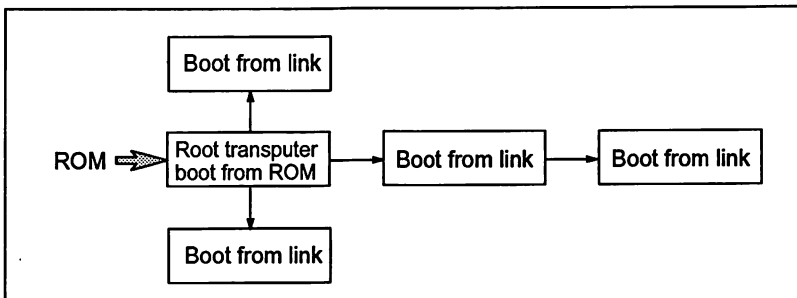


Figure 7.1 Loading a network from ROM

Some 32 bit transputers have a configurable external memory interface. For these transputers a memory configuration file may be created and put into ROM together with the application. A description of memory configuration files and how to create them is given in Chapter 6.

7.2 Prerequisites to using the `ieprom` tool

For an application file to be suitable for programming into ROM it must have been configured to be booted from ROM rather than booted from link. This selection is made by specifying the appropriate command line option when using the configurator and collector tools. See chapters 6 and 3 respectively. It is also essential that all C and FORTRAN programs, including those targeted at a single processor, are configured; programs prepared with the `icollect 'T'` option are not in a format suitable for `ieprom`.

7.3 Running `ieprom`

`ieprom` takes as input a control file and outputs one or more files which may be put into ROM by an EPROM programmer.

The control file, in text format, specifies the root transputer type, the name of the bootable file containing the application, the memory configuration file (if one is being used), the amount of space available in the EPROM, and the format that the output is to take. Available output formats are: binary, hex dump, Intel, Extended Intel, or Motorola S-Record format.

The `ieprom` tool is invoked by the following command line:

► `ieprom filename { options }`

where: *filename* is the name of the control file.

options is a list of options from Table 7.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

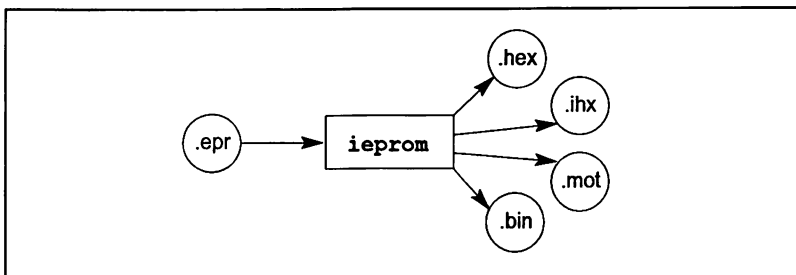
Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
I	Selects verbose mode. In this mode the user will receive status information about what the tool is doing during its operation, for example reading or writing to a file.
R	Directs <code>ieprom</code> to display the absolute address of the code reference point. This address can be used to locate within the memory map created by the <code>icollect 'P'</code> option.

Table 7.1 `ieprom` command line options

The operation of `ieprom` in terms of standard file extensions is shown below.



7.3.1 Examples of use

`ieprom` may be invoked in verbose mode by using one of the following commands:

`ieprom -i mycontrol.epr` (UNIX based toolsets)

`ieprom /i mycontrol.epr` (MS-DOS and VMS based toolsets)

7.4 ieprom control file

The control file is a standard text file, prepared with an editor; it consists of comments and statements. A comment is any blank line or any text following the comment marker `'--'`. Comments are ignored by the `ieprom` tool.

Statements are all other lines within the file. They may be in any order, except that the four statements defining a block must immediately follow the statement `'output.block'` (see table 7.3). Statements may be interspersed with comments.

Individual statements are constructed of a keyword and an associated parameter. These must be separated by at least one space or tab but extra spaces may be inserted before, between, or after them for aesthetic purposes. The statements are listed, along with their parameters, in tables 7.2 to 7.4.

Examples of control file contents are given in section 7.8.

The statements in table 7.2 are used to specify the contents of the EPROM: the processor type, the source of the data (code and memory configuration) to be placed in the EPROM, and the total size of EPROM memory.

Statement	Parameter/Description
root.processor.type	<p><i>type</i></p> <p>This statement specifies the processor type. The processor type can be specified in full (e.g. T400), or one of the following classes can be specified:</p> <p>T2: 16 bit processor (M212, T212, T222, T225)</p> <p>T4: 32 bit processor (T400, T414, T425, <i>not</i> T426)</p> <p>T8: 32 bit processor with FPU (T800, T801, T805)</p> <p>The IMS T426 <i>must</i> be specified as T426. See appendix B for a full list of valid processor types.</p> <p>This statement <i>must</i> be present as the first line in the control file.</p>
bootable.file	<p><i>filename</i></p> <p>This statement specifies the file that contains the output of <code>icollect</code>, usually the application plus its ROM loader(s).</p> <p>This statement <i>must</i> be present in the control file.</p>
memory.configuration	<p><i>filename</i></p> <p>This statement specifies a T4/T8 memory configuration file to be included in the EPROM image. This file is a standard memory configuration description (see chapter 6 for details).</p> <p>This statement is optional. If absent from the control file then no memory configuration will be inserted in the output file.</p>
eprom.space	<p><i>hex number</i></p> <p>This statement specifies the size of the EPROM memory space in bytes. This space may actually contain several physical devices.</p> <p>This statement <i>must</i> be present in the control file.</p>

Table 7.2 Specifying the EPROM contents

The statements in table 7.3 specify the output to be produced: the format of the data and whether the data is to be placed in a single file or split into blocks.

Statement	Parameter/Description								
output.format	<p>hex intel extintel srecord binary</p> <p>This statement specifies the output file format as being one of: plain ASCII hex, Intel hex, extended Intel hex, Motorola S-record or binary format respectively. These output formats are explained in section 7.6.</p> <p>This statement is optional. If absent from the control file then the default output is hex.</p>								
output.all	<p><i>filename</i></p>								
output.block									
	<p><i>filename</i></p> <p>These statements are used to specify the type of output and the output filename. By convention the following file extensions should be used:</p> <table> <tr> <td>.hex</td><td>Hexadecimal</td></tr> <tr> <td>.bin</td><td>Binary</td></tr> <tr> <td>.ihx</td><td>Intel formats</td></tr> <tr> <td>.mot</td><td>Motorola format</td></tr> </table> <p>output.all means that all of the image is to be output to one file.</p> <p>output.block specifies that a block of data is to be output to the specified file. It must be followed by the four statements that define the block; these are detailed in table 7.4.</p> <p>The control file <i>must</i> contain one output.all statement, or one or more output.block statements.</p>	.hex	Hexadecimal	.bin	Binary	.ihx	Intel formats	.mot	Motorola format
.hex	Hexadecimal								
.bin	Binary								
.ihx	Intel formats								
.mot	Motorola format								

Table 7.3 Specifying the output format

Table 7.4 lists the statements used to define each output block. One of each of these statements must follow each **output.block** statement.

Statement	Parameter/Description
start.offset	<i>hex number</i> This statement specifies the address of the start of the block, as a byte offset into the EPROM space.
end.offset	<i>hex number</i> This statement specifies the address of the end of the block, as a byte offset into the EPROM space.
byte.select	<i>byte list all</i> This statement is followed by either a list of byte numbers (separated by &), or the keyword all . It specifies which bytes in a word are to be output in this block. The byte numbers can be 0, 1, 2 and 3 for 32 bit processors; or 0 and 1 for 16 bit processors.
output.address	<i>hex number</i> This statement specifies the byte address, in the EPROM programmer's memory map, at which the block is to be output.

Table 7.4 Output block specification

7.5 What goes into the EPROM

This section describes the contents of the EPROM, the reasons behind the code layout and the function of those components inserted by **ieprom**.

The contents of the EPROM includes the bootable file, traceback data and jump instructions to enable the processor to find the start of the bootable file. Should the user define the memory configuration this information will also be placed in the EPROM. The general layout of the code in the EPROM is shown in figure 7.2.

7.5.1 Memory configuration data

Memory configuration data, when present, is placed immediately below the top word of the EPROM. The top word holds the first instructions to be executed if the transputer is booting from ROM.

If the processor has a configurable memory interface it will scan the memory configuration data held on the EPROM, before executing the first instructions. If a standard memory configuration is being used there should be no memory configuration data present and the processor will ignore this section of the EPROM.

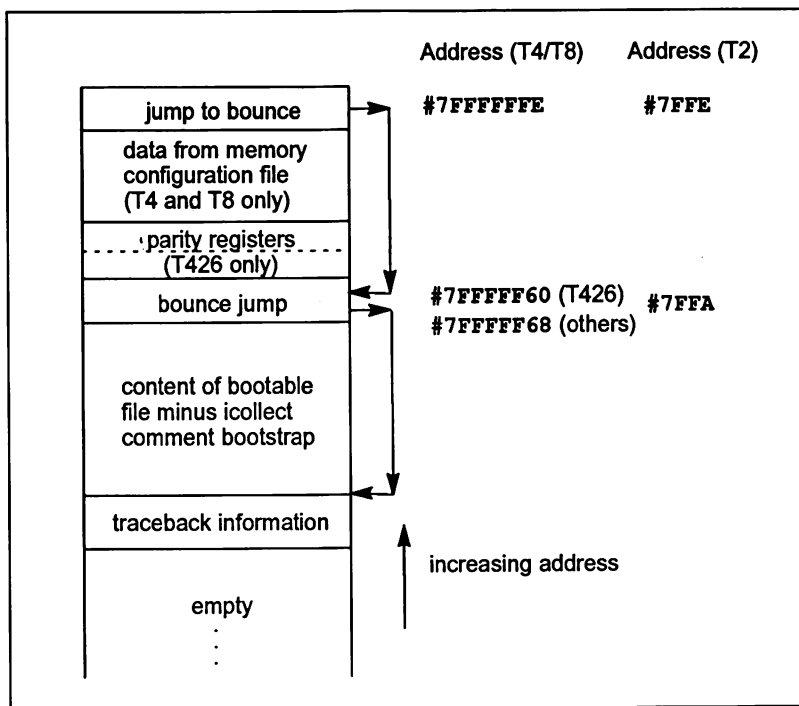


Figure 7.2 Layout of code in EPROM

7.5.2 Parity registers

The T426 has the `ParityErrorReg` and `ParityErrorAddressReg` mapped into the two words immediately below the memory configuration data (addresses #7FFFFFF64 and #7FFFFFF68). The EPROM tool needs to that it must avoid these addresses on the T426 and so the processor type must be given explicitly in the `root.processor.type` statement.

7.5.3 Jump instructions

The first instruction executed by the processor when booting from EPROM, is located at *most positive integer* – 1: this is #7FFFFFFE for 32-bit machines and #7FFE for 16-bit machines. The first two instructions cause a backwards jump to be made, with a distance of up to 256 bytes; however, since most applications are larger than 256 bytes it is necessary for `ieprom` to insert a 'bounce' jump back to the start of the bootable file.

7.5.4 Bootable file

The bootable file will have been produced by the collector tool `icollect`, using a boot from ROM loader. The comment `bootstrap`, containing traceback information originally added to this file by `icollect`, is stripped off by `ieprom`.

The bootable file is placed in the EPROM such that the start of the file is placed at the lowest address, with the rest of the file being loaded in increasing address locations. The end of the file is placed immediately below the bounce jump instruction, which points to the start of the bootable file.

7.5.5 Traceback information

`ieprom` creates its own traceback information consisting of the name of the control file and the time at which `ieprom` ran. This information is placed below the start of the bootable file. **Note:** at present this information is not used by any of the tools.

7.6 ieprom output files

The tool can produce output in a form readable by the user or in a form readable by EPROM programming devices. The following formats are supported:

- Binary output
- Hex dump
- Intel hex format
- Intel extended hex format
- Motorola S-record format

Whichever form is used, it is sometimes necessary to output the data in separate blocks. Block mode operation is discussed in section 7.7.

Note: there is no output for unused areas of the EPROM. If the buffer in the EPROM programmer is not initialized before loading the files produced by this program into it, unused areas of the EPROM will be filled with random data. Although the operation of the bootstrap code and loader programs will not be affected by the presence of random data, these areas of the EPROM cannot subsequently be programmed without erasing the whole device.

7.6.1 Binary output

This file is in binary format and simply contains all bytes output. There is no additional information such as address or checksums.

7.6.2 Hex dump

This simple format is intended to be used to check the output from the program. The dump consists of rows of 16 bytes each, prefixed by the address of the first

byte of each row. The format contains no characters other than the hexadecimal digits, the space character and newlines.

7.6.3 Intel hex format

This is a commonly used protocol for EPROM programming equipment. A sequence of data records is sent. Each record contains a few bytes of information, a start address and a checksum. In addition, a special record marks the end of a transmission. Since the format only supports 16-bit addresses, any longer addresses will generate an error message. Records produced by this program contain at most 32 bytes each.

7.6.4 Intel extended hex format

This format, also known as Intel 86 format, is similar to Intel hex, but adds another type of record. The new type 02 record is used to specify addresses of more than 16 bits. The type 02 record contains a 16-bit field giving a segment base offset. This value is shifted left four places and added to subsequent addresses. This mimics the operation of the segment registers on the Intel 8086 range of microprocessors. The segment base offset value persists until the next type 02 record occurs. This format therefore allows addresses up to 20 bits in length. Again, longer addresses will generate an error message. The program minimizes the number of type 02 records inserted in its output.

7.6.5 Motorola S-record format

This format is another well known industry standard; it consists of a header record, data records, and finally an image end record. The advantage of this format is that, by the use of different data record types, it can support 16, 24, or 32 bit addresses. This program uses whichever data record type is necessary.

7.7 Block mode

Block mode is a term used to describe the output from ieprom, when more than one output file is produced. The user defines how the data is to be split between files using control file statements (see table 7.4).

7.7.1 Memory organization

In order to understand the ideas behind block mode operation it is helpful to understand the way memory is organized in a 16 or 32 bit transputer.

In general, a transputer with a 32 bit data bus will expect to read from memory in 32 bit words; the addresses of these words will be on word boundaries (i.e. the address will always be divisible by 4, the two least significant bits will be 0). EPROM

devices, however, are usually 8 bits wide, and so it is necessary to have 4 EPROMs side by side to make up the 32 bit width. These 4 devices are addressed as bytes 0 to 3. The two least significant bits of an address (the 'byte selector') give the byte numbers.

Similarly a 16 bit transputer will expect to read from memory in 16 bit words. The address of each word will always be divisible by 2. The two EPROM devices required to make up the 16 bit width will be addressed as bytes 0 and 1. In this case the least significant bit of an address indicates the byte being accessed.

7.7.2 When to use block mode

Block mode has three uses:

- When the EPROM programmer being used is unable to split the input data into bytes, in order to program separate byte wide devices.
- When the EPROM programmer has insufficient memory to hold the entire image.
- When it is necessary, for some reason, to load the program to a different address in the EPROM programmer to that which it will occupy in the EPROM space.

7.7.3 How to use block mode

When block mode is to be used, the user must first decide on the blocks to be output. For each block an `output.block` statement must be specified in the control file. Each `output.block` statement must be followed by the four statements:

```
start.offset  
end.offset  
byte.select  
output.address
```

`ieprom` will scan the entire image and output those bytes that have an EPROM space address between `start.offset` and `end.offset` and whose byte address matches the `byte.select` value. It will output this data to contiguous addresses starting at `output.address`.

Note: if the image does not occupy all of the EPROM space then there may be some space at `output.address` before the data starts.

7.8 Example control files

7.8.1 Simple output

For this example the application is in the file `bootable.btr`, there is no memory configuration, there is 128 kbytes of EPROM, and the EPROM programmer can take all of the code as one file.

```
-- EPROM description file for example 1
root.processor.type  T4
bootable.file       bootable.btr
eprom.space         20000
output.format       srecord
output.all          image.mot
```

7.8.2 Using block mode

In this example the application is in `embedded.btr`, there is a memory configuration in `fastsram.mem`, there are 16 kbytes of EPROM and the data is to be split into four blocks of 4k EPROMs to be programmed at locations 0000, 1000, 2000, and 3000 in the EPROM programmer's memory.

```
-- EPROM description file example 2
```

```
root.processor.type  T8
bootable.file       embedded.btr
memory.configuration fastsram.mem
eprom.space         4000
output.format       intel
```

```
output.block        part1.ihx
  start.offset      0000
  end.offset        3FFF
  byte.select       0
  output.address    0000
```

```
output.block        part2.ihx
  start.offset      0000
  end.offset        3FFF
  byte.select       1
  output.address    1000
```

```
output.block        part3.ihx
  start.offset      0000
  end.offset        3FFF
  byte.select       2
  output.address    2000
```

```
output.block        part4.ihx
  start.offset      0000
  end.offset        3FFF
  byte.select       3
  output.address    3000
```

7.9 Error and warning messages

The following is a list of error and warning messages the tool can produce:

Command line parsing error

This indicates that a command line option has been specified that the tool does not recognize.

No input file specified

This indicates that when trying to invoke the tool the user has not specified a control file to use as input.

Unable to open control file '*filename*'

The control file specified cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open configuration file '*filename*'

The memory configuration file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open bootable file '*filename*'

The bootable file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open output file '*filename*'

An output filename has been specified incorrectly. Check the format of the filename.

Control file error

This message will be received whenever an error is found in the format of the control file. A self explanatory message will be appended, giving details of what the tool expects the format to be.

8 `ilibr` — librarian

This chapter describes the librarian tool `ilibr` that integrates a group of compiled code files into a single unit that can be referenced by a program. The chapter begins by describing the command line syntax, goes on to describe some aspects of toolset libraries, and ends with some hints about how to build efficient libraries from separate modules.

8.1 Introduction

The librarian builds libraries from one or more separately compiled units supplied as input files. The input files may be any of the following:

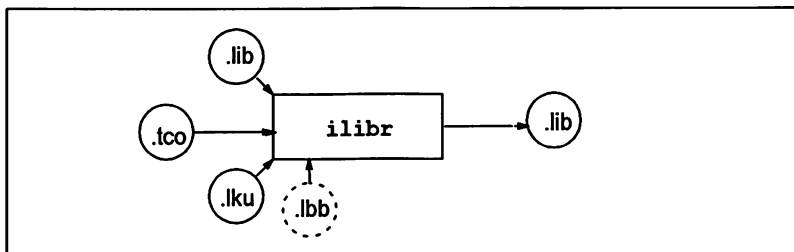
- Compiled object code files produced by the INMOS compilers:
 - `oc` (Occam 2 compiler),
 - `icc` (ANSI C compiler),
 - `if77` (FORTRAN-77 compiler).
- Library files already generated by `ilibr` (see section 8.2.4).
- Linked object files (see section 8.2.3).

The librarian takes a list of compiled files in TCOFF format and integrates them into a single object file that can be used by a program or program module. Each module in the input list becomes a selectively loadable module in the library. Input files can either be specified as a list on the command line or in *indirect files*.

The library, once built, will contain an index followed by the concatenated modules. The index is generated and sorted by the librarian to facilitate rapid access of the library content by the other tools in the toolset, for example, the linker.

Compiled object files (excluding library files) may be concatenated for convenience before using the librarian. This may prove useful when dealing with a large number of input files.

The operation of the librarian in terms of standard file extensions is shown below.



8.2 Running the librarian

To invoke the librarian use the following command line:

► **ilibr** *filenames* { *options* }

where: *filenames* is a list of input files separated by spaces.

options is a list of one or more options from Table 8.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

The number of file names allowed on a command line is system dependent. To avoid overflow, files may be concatenated or an indirect file used. It is the user's responsibility to ensure that the concatenation process does not corrupt the modules, for example by omitting to specify that the concatenation is to be done in binary mode.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
F <i>filename</i>	Specifies a library indirect file.
I	Displays progress information as the library is built.
O <i>filename</i>	Specifies an output file. If no output file is specified the name is taken from the first input file and a .lib extension is added.

Table 8.1 **ilibr** command line options

Example

```
ilibr myprog.t4x myprog.t8x
```

In this example, the compiled object code files **myprog.t4x** and **myprog.t8x** (compiled for T4 and T8 transputers respectively) are used to create a library. Because no output file name is specified on the command line, the library will be given the name **myprog.lib**.

8.2.1 Default command line

A set of default command line options can be defined for the tool using the **ILIBRARG** environment variable. Options must be specified in the variable using the syntax required by the command line.

8.2.2 Library indirect files

Library indirect files are text files that contain lists of input files, directives to the librarian, and comments. Filenames and directives must appear on different lines. Comments must be preceded by the double dash character sequence '---', which causes the rest of the line to be ignored. By convention indirect files are given the .libb extension.

Indirect files may be nested within each other, to any level. This is achieved by using the #INCLUDE directive. By convention nested indirect files are also given the extension .libb.

The following is an example of an indirect file:

-- user's .libb file

```
userproc1.tco      -- single modules
userproc2.tco
userproc3.tco
myconcat.tco       -- concatenation of modules
#include indirect.libb -- another indirect file
userproc4.tco      -- another single module
```

The contents of a nested indirect file will effectively be expanded at the position it occurred.

To specify indirect files on the command line each indirect filename must be preceded by the 'F' option.

8.2.3 Linked object input files

The librarian will also accept linked object files as input, with certain conditions. The facility to create libraries of linked modules provides an easy method of specifying input to the configurer. Such library files should only be referenced from a configuration description.

The librarian will generate an error if an attempt is made to include both linked units and compiled modules in a single library. In addition, libraries of linked object modules must *not* be used as input to the linker ilink. This is because the linker does not accept linked units as input files.

8.2.4 Library files as input

Library files can themselves be used as input files to ilibr. When a library file is used as a component of a new library, its index is discarded by ilibr.

Library files may not be concatenated for input to the librarian.

8.3 Library modules

Libraries are made up of one or more selectively loadable modules. A module is the smallest unit of a library that can be loaded separately. Modules are selected via the library index.

8.3.1 Selective loading

Libraries can contain the same routines compiled for different transputer types and (for occam modules) in different error modes.

Selection of library modules for linking in with the program is made on the basis of target processor type and error mode. For example, if the program is compiled for an IMS T414 only modules compiled for this processor type or for processors in a compatible transputer class are loaded. For languages such as FORTRAN and C the error mode is always UNIVERSAL.

For C and FORTRAN modules the linker selects the library modules best suited to the compilation units. For occam the compiler identifies the modules to be selected according to the requirements of the main program. The linker then makes the selection.

The linker also selects library modules for linking on the basis of usage. Only those modules that are actually used by the program are linked into the program.

8.3.2 How the librarian sorts the library index

The librarian creates a library index which is used by the linker to select the required modules. The librarian sorts the index so that for a given processor type, the optimum module is always selected by the linker.

The librarian compares and sorts modules according to a number of factors including attributes set by the compiler options used. These determine for example, the instruction set of the module and influence run-time execution times.

For example, where two library modules were derived from the same source but compiled for classes TA and T4, the librarian would place the T4 module first because it uses a larger instruction set. Modules compiled with interactive debugging enabled are placed later in the index than those for which debugging is disabled. The librarian orders the index entries such that the first valid entry is always the 'best choice'. If two entries are found to be identical the librarian will issue a warning.

8.4 Library usage files

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They consist of a list of separately compiled units or libraries referenced within a particular library. The .liu files required by the tool-set's libraries are supplied by INMOS.

If the `imakef` tool is used then library usage files should be created for all libraries that are supplied without source. This is to enable the `imakef` tool to generate the necessary commands for linking. Library usage files are text files. They may be created for a specific library by invoking the `imakef` tool and specifying a .liu target. See section 11.5.

Such files are given the same name as the library file to which they relate but with an `.liu` extension.

8.5 Building libraries

This section describes the rules that govern the construction of libraries and contains some hints for building and optimizing libraries.

8.5.1 Rules for constructing libraries

1. Routines of the same name in a library must be compiled for different transputer types, error modes or debug attributes.
2. Libraries that contain modules compiled for a transputer class (i.e. TA or TB) are treated as though they contain a copy for each member of the class.
3. Libraries that contain modules compiled in UNIVERSAL mode are treated as though they contain a copy for each of the two error modes HALT and STOP.
4. Libraries that contain modules with interactive debugging enabled are treated as though they also contain a copy with interactive debugging disabled. (When interactive debugging is enabled, channel input/output is performed via library calls otherwise transputer instructions are used).

8.5.2 General hints for building libraries

Routines that are likely to be used together in a program or procedure (such as routines for accessing the file system) can be incorporated into the same library. At a lower level, routines that will *always* be used together (such as those for opening and closing files) can be incorporated into the same module.

Libraries can contain the same routines compiled for different transputer types, in different error modes and with different input/output access to channels. Only those modules actually used by the program are incorporated by the compiler and linked in by the linker.

Where possible compile library input files with debugging enabled. This enables the debugger to locate the library source if an error occurs inside the library.

When building C libraries care should be taken if the 'FS' or 'FC' INMOS C compiler command line options are used, that code compatibility is maintained.

8.5.3 Optimizing libraries

It is possible for the user to optimize the size and content of any libraries which he builds himself, to target appropriate processors, improve the speed of code execution and to provide the best code for a given processor.

All libraries

Points to consider when constructing libraries in any language or mixture of languages:

- Whether the library is to be targeted at one or two specific processors or a wide range of processors. The transputer type specified for the compilation of a library module determines the instruction set used. Transputer classes TA and TB provide the basic instruction sets common to several transputer types. Transputer classes such as the T5 provide extended instruction sets but are targetted at fewer transputers than classes TA and TB.
- For floating point operations, classes T5 and TB provide better code and therefore better execution times than class TA.
- Whether the versatility of the library should be reduced in order to create a smaller library.

Libraries containing occam modules

When building libraries which include modules written in OCCAM the same considerations apply, but also note the following:

- The error mode used will affect the size of the library. A library created from modules compiled in UNIVERSAL mode will behave as if it contains a copy of the code for both HALT and STOP mode. Also, on the current range of transputers, code compiled in HALT mode will tend to execute faster than if it is compiled in STOP or UNIVERSAL error modes.
- For libraries containing modules where the method of channel input/output may be altered, (such as in occam), both the availability of the interactive debugging facility and the speed at which the code will be executed may be affected.

When interactive debugging is enabled, channel input/output will be implemented via library calls. When interactive debugging is disabled using the compiler 'x' option, transputer instructions are used for channel input/output. This leads to faster execution times. However, disabling interactive debugging for one module of a program, will disable this facility for the whole program.

For a detailed description of transputer types and error modes, see appendix B.

Outlined below are three different approaches to optimization. The first approach provides the greatest level of flexibility in its application. The experienced user may refine these guidelines to specific requirements.

Semi-optimized library build targeted at all transputer types

This is the simplest way to build a library that covers the full range of transputers.

The user should compile each module separately for the following three cases and incorporate all three versions into the library.

Processor type/class	Error mode	Method of channel I/O
T2	UNIVERSAL	Via library calls
TA	UNIVERSAL	Via library calls.
T8	UNIVERSAL	Via library calls.
Note: Error mode and channel i/o only apply only to modules which employ them e.g. OCCAM modules compiled by oc.		

The resulting library will be small in terms of the number of modules it will contain. Due to their generic nature the modules themselves may be bulky and because they contain only the base set of instructions, the execution time for the program will tend to be slower than a more optimized approach.

Optimized library targeted at all transputer types

In order to build a library which is both generalized enough to work for all 32-bit transputers and is then optimized for modules which require extended instructions sets the following approach is recommended:

1. Compile all modules for classes TA and T8. This will provide modules which can be run on all 32-bit transputers.
2. If any of the modules perform floating point operations, compile these modules for class TB as well.

For 16-bit transputers it should be sufficient to compile all modules for class T2.

Library build targeted at specific transputer types

This method of building a library will limit the use of the library modules to specific transputer types and error modes. It is recommended as the simplest strategy to use when the following options are known for each module:

- Target transputer type.
- Error mode of modules, if any, (i.e. HALT, STOP or UNIVERSAL).
- Method of channel input/output, if any.

All modules to be included in the library must be compiled for *each* target transputer type and, if appropriate, for the same error mode and method of channel input/output. The resulting library may be large and contain a certain amount of duplication.

For example, for the following options:

- T414 and T425 processor types
- HALT error mode
- channel input/output via library calls

each module should be compiled for the following:

Processor type/class	Error mode	Method of channel I/O
T414	HALT	Via library calls
T425	HALT	Via library calls.
Note: Error mode and channel i/o only apply only to modules which employ them e.g. OCCAM modules compiled by oc.		

8.6 Error Messages

This section lists each error and warning message which may be generated by the librarian. Messages are in the standard toolset format which is explained in appendix A.

8.6.1 Warning messages

filename - bad format: symbol *symbol* multiply exported

An identical symbol has occurred in the same file. There are three possibilities:

The same file has been specified twice.

The file was a library where previous warnings have been ignored.

A module in the file has been incorrectly generated.

filename1 - symbol *symbol* also exported by *filename2*

An identical symbol has occurred in more than one module. If the linker requires this symbol, it will never load the second module.

8.6.2 Serious errors

bad format: *reason*

A module has been supplied to the librarian which does not conform to a recognized INMOS file format or has been corrupted.

filename - line number - bad format: excessively long line in indirect file

A line is too long. The length is implementation dependent, but on all currently supported hosts, is long enough to only be exceeded in error.

filename - line number - bad format: file name missing after directive

A directive (such as `INCLUDE`) has no file name as an argument.

filename - line number - bad format: non ASCII character in indirect file

The indirect file contains some non printable text. A common mistake is to specify a library or module with the `F` command line argument or the `INCLUDE` directive.

bad format: not a TCOFF file

The supplied file is not a library or module of any known type.

filename - line number - bad format: only single parameter for directive

The directive has been given too many parameters.

command line error token

An unrecognized token was found on the command line.

filename - could not open for reading

The named file could not be found/opened for reading.

filename1 - line number - could not open filename2 for reading

The file name specified in an `INCLUDE` directive could not be opened.

filename - could not open for writing

The named file could not be opened for writing.

filename - must not mix linked and linkable files

The librarian is capable of creating libraries from compiled modules or linked units, but it is illegal to attempt to create a library from both.

no files supplied

Options have been given to the librarian but no modules or libraries.

filename - nothing of importance in file

The file name specified in a library indirect file or in an `INCLUDE` directive was empty or contained nothing but white space or comments.

filename - line number - only one file name per line

More than one file name has been placed on a single line within an indirect file.

*filename - line number - **unrecognised directive** directive*

An unrecognized directive has been found in an indirect file.

9 `ilink` — linker

This chapter describes the linker tool `ilink` which combines a number of compiled modules and libraries into a linked object file. The chapter begins with a short introduction to the linker, explains the command line syntax and goes on to describe linker indirect files and the main linker options. The chapter ends with a list of linker messages.

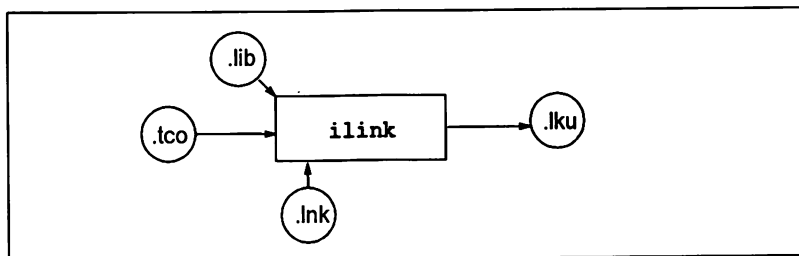
9.1 Introduction

The linker links a number of compiled modules and library files into a single linked object file (known as a linked unit), resolving all external references. The linker may be used to link object files produced by the ANSI C compiler `icc`, the occam 2 compiler `oc`, and the FORTRAN-77 compiler `if77`. Code produced by the linker can be used as input to the configurator and collector tools to produce a bootable code file.

The linker can be driven directly from the command line or indirectly from a *linker indirect file*. This is a text file which contains a list of files to be linked, together with directives to the linker.

The linker is designed to accept input files in the Transputer Common Object File Format (TCOFF) supported by this release of the toolset. However, the linker can be directed to produce output files in Linker File Format (LFF). In this format the output is compatible with either the `iboot` or `iconf` tools used by previous² INMOS toolset releases.

The operation of the linker in terms of standard toolset file extensions is shown below.



2. Pre-TCOFF toolsets, for example the Dx05 occam toolsets.

9.2 Running the linker

To invoke the linker use the following command line:

► **ilink** [*filenames*] {*options*}

where: *filenames* is a list of compiled files or library files.

options is a list of the options given in Table 9.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

If an error occurs during the linking operation no output files are produced.

Example of use:

UNIX based toolsets:

```
icc hello
ilink hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

In this example a compiled C file is linked for the default T414 transputer, using the standard C startup linker indirect file *cstartup.lnk*. The example also shows the steps for compiling, booting and loading the program.

Option	Description
<i>Transputer type</i>	See appendix B for a list of options to specify transputer type.
EX	Allows the extraction of modules without linking them.
F filename	Specifies a linker indirect file.
H	Generates the linked unit in HALT mode. This is the default mode for the linker and may be omitted for HALT mode programs. This option is mutually exclusive with the 'S' option.
I	Displays progress information as the linking proceeds.
KB memorysize	Specifies virtual memory required in Kilobytes.
LB	Specifies that the output is to be generated in LFF format, for use with the <code>iboot</code> bootstrap tool and <code>iconf</code> configurer tool used in earlier INMOS toolsets. (See footnote 2).
LC	Specifies that the output is to be generated in LFF format, for use with the <code>iconf</code> tool used in earlier INMOS toolsets. (See footnote 2).
ME entryname	Specifies the name of the main entry point of the program and is equivalent to the <code>#mainentry</code> linker directive (See below).
MO filename	Generates a module information file with the specified name.
O filename	Specifies an output file.
S	Generates the linked unit in STOP mode. This option is mutually exclusive with the 'H' option.
T	Specifies that the output is to be generated in TCOFF format. This format is the default format.
U	Allows unresolved references.
X	Generates the linked unit in UNIVERSAL error mode, which can be mixed with HALT and STOP modes.
Y	Disables interactive debugging for occam code. Used when linking in occam modules compiled with interactive debugging disabled.

Table 9.1 `ilink` command line options

9.2.1 Default command line

A set of default command line options can be defined for the tool using the `ILINKARG` environment variable. Options must be specified using the syntax required by the command line.

9.3 Linker indirect files

Linker indirect files are text files containing lists of input files and commands to the linker. Indirect files are specified on the command line using the 'F' option.

Linker indirect files can contain filenames, linker directives, and comments. Filenames and directives must be on separate lines. Comment lines are introduced

by the double dash ('--') character sequence and extend to the end of line. Comments must occupy a single line.

Indirect files can include other indirect files.

Linker indirect files must be created for all link operations which involve the use of `imakef` and either C or FORTRAN modules. For further details see section 11.4.

9.3.1 Linker indirect files supplied with the toolset

Linker indirect files supplied with the toolset are described in section 3.11 of the *Toolset User Guide*. The purpose of these files is to reference various runtime libraries (or in the case of occam, compiler libraries) required to link application programs. When specifying the program modules to be linked, the appropriate linker indirect file must be included on the linker command line.

9.4 Linker directives

The linker supports six directives which can be used to fine tune the linking operation. Linker directives must be incorporated in indirect files (they cannot be specified on the linker command line) and are introduced by the hash ('#') character.

The six linker directives are summarized below and described in detail in the following sections.

Directive	Description
#alias	Defines a set of aliases for a symbol name.
#define	Assigns an integer value to a symbol name.
#include	Specifies a linker indirect file.
#mainentry	Defines the program main entry point.
#reference	Creates a reference to a given name.
#section	Defines the linking priority of a module.
Note: Symbol names are case sensitive.	

9.4.1 `#alias basename {aliases}`

The `#alias` directive defines a list of aliases for a given base name. Any reference to the alias is converted to the base name before the name is resolved or defined. For example, if a module contains a call to routine `proc_a`, which does not exist, then another routine `proc_d` may be given the alias `proc_a` in order to force the call to be made to routine `proc_d`.

```
#alias proc_d proc_a
```

In the above example the reference to `proc_a` is considered to be resolved. Modules may be loaded from the library for `proc_d` but the linker will not attempt to

search for library modules for `proc_a`. If a procedure called `proc_a` is found in any module then an error will result as the symbol will be multiply defined.

9.4.2 `#define symbolname value`

The `#define` directive defines a symbol and gives it a value. This value must either be an optionally signed decimal integer, or an unsigned hexadecimal integer. (If it is the latter it must be preceded by a `#` sign). `#define` is also discussed in section 9.5.4.

Note: this directive is not applicable to `occam`.

9.4.3 `#include filename`

The `#include` directive allows a further linker indirect file to be specified. Linker indirect files can be nested to any level. The following is an example of nested indirect files:

-- user's .lnk file:

```
userproc1.tco      -- module
#mainentry proc_a  -- main entry point directive
#include sub.lnk    -- nested indirect file
```

-- user's sub.lnk file:

```
userproc2.tco      -- further modules
userproc3.tco
userlib.lib         -- library
```

9.4.4 `#mainentry symbolname`

The `#mainentry` directive defines the main entry point of the program i.e. the top level function of the program. This directive is equivalent to the `'ME'` command line option. Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default. If there is more than one such symbol the linker will warn that there is an ambiguity.

For C and FORTRAN programs the supplied linker indirect startup files define the system main entry point.

9.4.5 `#reference symbolname`

The `#reference` directive creates a forward reference to a given symbol. This allows names to be made known to the linker in advance, or forces linking of library modules that would otherwise be ignored. The purpose is to allow the inclusion of library initialization routines which might not otherwise be included. For example:

```
#reference open
```

The above example causes `open` to be included in the link, whether it is needed or not.

9.4.6 `#section name`

The `#section` enables the user to define the order in which particular modules occur in the executable code.

In order to use this directive the program modules must have been compiled using the compiler pragma `IMS_linkage` (C programs) or `LINKAGE` (Occam programs). Details of the appropriate directive can be found within the compiler reference chapter of this manual.

A compiler directive enables a section name to be associated with the code of a compilation module. A section name may take the default value `"pri%text%base"` or a name specified by the user.

The linker will place modules associated with the section name `"pri%text%base"` first in the code of the linked unit, in the order in which these modules are encountered. When the linker directive `#section` is used this default condition is overridden. The modules identified by user defined section names will be placed first in the linked module, in the order in which the `#section` directives are encountered. These will be followed by any other modules in an undefined order at the end of the linked unit.

For example:

```
#section first%section%name
#section second%section%name
```

In the above example any modules identified by `first%section%name` will be linked first, followed by modules identified by `second%section%name`, followed by any other modules.

9.5 Linker options

9.5.1 Processor types

A number of options are provided to enable the user to specify the target processor for the linked object file, see appendix B. Only one target processor or transputer class may be specified and this must be compatible with the processor types or transputer class used to compile the modules.

If no target processor is specified, the processor type for the linked object file will default to a T414 processor type.

If any input file in the list is incompatible with the processor type in use, the link fails and an error is reported.

9.5.2 Error modes – options H, S and X

Linked code may be generated in three error modes. For C or FORTRAN modules, compiled respectively using `icc` or `if77`, the error mode will be UNIVERSAL. OCCAM modules, compiled by `oc`, may be compiled in one of three error modes as shown in table 9.2.

Error mode	Description
HALT	An error halts the transputer immediately.
STOP	An error stops the process and causes graceful degradation.
UNIVERSAL	Modules compiled in this mode may be run in either HALT or STOP mode depending on which mode is selected at link time.

Table 9.2 Error modes

Modules that are to be linked together must be compiled for compatible error modes. C and FORTRAN modules can be mixed with OCCAM modules and OCCAM modules compiled for different error modes may also be mixed. Table 9.3 indicates the compilation error modes which are compatible and the possible error modes they may be linked in.

Compatible error modes	ilink options
HALT, UNIVERSAL	H
STOP, UNIVERSAL	S

Table 9.3 ilink error modes

Note: Modules which have been compiled in UNIVERSAL error mode may be *linked* in this mode using the X option. If the resulting linked unit is then processed by the `icollect` tool it will be treated as if it had been linked in HALT mode.

The linker will produce an error if an input file is in a mode incompatible with the command line options or defaults. The linker default is to create linked modules in HALT mode unless otherwise specified.

9.5.3 TCOFF and LFF output files – options T, LB, LC

These three options enable the format of the linked unit output file to be changed. The linker will default to TCOFF output if none is specified.

Option **T** specifies that the linked unit is to be output in TCOFF format. This file may then be processed normally by other tools in the toolset, for example, the configurator and collector tools.

The **LB** and **LC** options specify that the linked unit is to be output in LFF format so that it is compatible with previous toolsets. The **LB** option produces a file compat-

ible with the `iboot` and `iconf` tools used by earlier INMOS toolsets. (See footnote 2). The specified main entry point of the linked program is then available for bootstrapping by `iboot` or configuring by `iconf`. The `LC` option is used only in mixed language systems incorporating OCCAM programs. No main entry point need be specified.

When the `LB` and `LC` options are used the linked output file will not be compatible with the current toolset, which requires TCOFF format.

9.5.4 Extraction of library modules – option `EX`

The `EX` option instructs the linker to extract the modules which would normally have been linked by the `ilink` command, and to insert them unmodified into an output file. When the `EX` option is used, the linker does not produce a linked unit as output. Instead it outputs a concatenation of the component modules that would have made up the linked unit. This file can then itself be used as input to either the linker or librarian. By default the output file produced will have the extension `.1ku`, although it is not a linked unit. An alternative output filename and extension can be specified using the `ilink O` option.

This mechanism can be used for creating sub units for linking at a later date or for extraction of modules from libraries.

When linking or extracting modules the linker attempts to resolve any unresolved references. The linker `U` option and the `#reference` directive are particularly useful for controlling the extraction of unlinked modules. For non-OCCAM modules the `#define` directive can also be used to refine the selection of modules which are extracted. Linker options and directives used in conjunction with the `EX` option do not modify the extracted modules, they just influence the selection process.

Example: Extraction from a user library

This example demonstrates how to extract sub-parts of a previously supplied library.

Suppose we are given a library, `mylib.lib`, which contains routines with entry-points `start`, `run`, `clear`, and `stop`. These routines may also call other modules which reside in the same library, but we are not concerned about their exact names. We can use the linker's `EX` option to extract a sub-library, which just contains `start`, `run`, and `stop`, but does not contain `clear`.

We do this by forcing the linker to 'find' references to `start`, `run` and `stop`, but leave out `clear`.

- 1 Create the following linker indirect file `x.lnk`:

```

— Items wanted
#reference start
#reference stop
#reference run

— Libraries
mylib.lib
```

- 2 Use `ilink` to extract the required modules and place them in a named file:

```
ilink -f x.lnk -o sublib.tco -ex (UNIX)
ilink /f x.lnk /o sublib.tco /ex (MS-DOS and VMS)
```

This command will create a file called `sublib.tco` which will contain all the submodules required.

- 3 The librarian can then be used to create a library:

```
ilibr sublib.tco -o sublib.lib (UNIX)
ilibr sublib.tco /o sublib.lib (MS-DOS and VMS)
```

Example: Extraction from a user library, using the run-time library

The example demonstrates how to extract sub-parts of a previously supplied library which uses the run-time library.

Consider the same example as that described above, but where the routines `start`, `stop` and `run` have calls to the run-time library embedded inside them. We have to tell the linker not to complain about these references, because they will be resolved later, when `sublib.lib` is used.

- 1 We do the same as before, but we tell the linker not to complain about unresolved references, by using the `U` command line flag:

```
ilink -f x.lnk -o sublib.tco -ex -u (UNIX)
ilink /f x.lnk /o sublib.tco /ex /u (MS-DOS and VMS)
```

- 2 `sublib.tco` then be supplied to the librarian in the same way as before.

Example: Extraction from a user library, for multiple processor types

Suppose we are supplied with `mylib.lib` which contains the routines `start`, `stop`, `run`, and `clear` for both T400 and TA, and that we wish to create a library `sublib.lib` which contains everything except `clear`.

- 1 We use the same method as the first example to extract the T400 code:

```
ilink -f x.lnk -o sublib.t4 -ex -t400 (UNIX)
ilink /f x.lnk /o sublib.t4 /ex /t400 (MS-DOS and VMS)
```

This command will create a file called `sublib.t4` which will contain all the submodules compiled for T400.

- 2 We do the same again for TA:

```
ilink -f x.lnk -o sublib.ta -ex -ta (UNIX)
ilink /f x.lnk /o sublib.ta /ex /ta (MS-DOS and VMS)
```

This command will create a file called `sublib.ta` which will contain all the submodules compiled for TA.

- 3 The librarian can then be used to create a library containing both:

```
ilibr sublib.t4 sublib.ta -o sublib.lib (UNIX)
ilibr sublib.t4 sublib.ta /o sublib.lib (MS-DOS and VMS)
```

Example: Generation of a completely linkable library

Suppose we have built a library `mylib.lib`, which requires access to the run-time library, and we wish to supply this to another person, without having to supply the run-time library separately. We can arrange for the linker to extract all the required parts of the run-time library and add them to `mylib.lib`.

- 1 Create a linker indirect file `x.lnk` which contains `#reference` lines for each symbol in `mylib.lib`:

```
-- Items wanted
#reference start
#reference stop
#reference run

-- Libraries
mylib.lib

-- Linker indirect file to access run-time library
#include occam.lnk
```

The run-time library line should be adjusted depending on the type of processor which is being used:

Language	When	Linker indirect file†
C	Full run-time library	<code>clibs.lnk</code>
C	Reduced run-time library	<code>clibsrld.lnk</code>
occam	32-bit processors without an FPU	<code>occam.lnk</code>
occam	32-bit processors with an FPU	<code>occam8.lnk</code>
occam	16-bit processors	<code>occam2.lnk</code>
† The C linker indirect files apply to the Dx314 toolsets and the occam indirect files to the Dx305 toolsets.		

- 2 Use `ilink` to extract the required modules and place them in a named file:

```
ilink -f x.lnk -o fulllib.tco -ex (UNIX)
ilink /f x.lnk /o fulllib.tco /ex (MS-DOS and VMS)
```

This command will create a file called `fulllib.tco` which will contain all the submodules required.

- 3 The librarian can then be used to create the extended library `fulllib.lib` which will contain the user library together with any routines which are required from the run-time library.

```
ilibr fulllib.tco -o fulllib.lib (UNIX)
ilibr fulllib.tco /o fulllib.lib (MS-DOS and VMS)
```

Extraction using `#define`

A module is the smallest unit the linker can extract from a library, and a module may contain several functions. It is quite likely that a module contains functions which

are not required as well as functions which are referenced from modules which are required. To prevent a function from being extracted it is assigned a *dummy* value within a `#define` directive; any value will do. This causes any reference to it to be satisfied.

When the linker encounters a reference to a required function it will extract the whole module. However, if the module contains a function already specified in a `#define` directive, the function will be multiply defined and the linker will abort the extraction. It may be wise when a function is not required, to define all functions which are exported from that module, to some dummy value, thereby preventing them all from being extracted.

9.5.5 Display information – option **I**

This option enables the display of linkage information as the link operation proceeds.

9.5.6 Virtual memory – option **KB**

The **KB** option allows the user to specify how much memory the linker will use for storing the image of the users program. By default the linker will attempt to store the entire image in memory. In situations where memory is limited, an amount (≥ 1 Kbytes) may be specified. If the program is larger than the amount specified then the linker will use the host filing system as an intermediate store. A reduction in speed may be expected at link time.

9.5.7 Main entry point – option **ME**

The **ME** option defines the main entry point of the program i.e. the point from which linking will start. This option is equivalent to the `#mainentry` directive and takes as its argument a symbol name which is case sensitive.

Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default. If there is more than one such symbol the linker will warn that there is an ambiguity.

9.5.8 Link map filename – option **MO**

This option causes a link map file to be produced with the specified name. A file extension should be specified as there is no default available. If the option is not specified a separate link map file is not produced.

A link map file is a text file containing information about the position of modules in the code file.

9.5.9 Linked unit output file – **O**

The name of the linked unit output file can be specified using the **O** option. If the option is not specified the output file is named after the first input file given on the

command line and a `.1ku` extension is added. If the first file on the command line is an indirect file the output file takes the name of the first file listed in the indirect file.

Note: Because there is no restriction on the order in which files may be listed it is up to the user to ensure that the output file is named appropriately.

9.5.10 Permit unresolved references – option `U`

The linker normally attempts to resolve all external references in the list of input files and reports any that are unresolved as errors.

Sometimes it is desirable to allow unresolved external references, for example during program development. The `U` option allows the link to proceed to completion by assuming unresolved references are to be resolved as zero. Warning messages may still be generated and the program will only execute correctly if such references are in fact redundant.

9.5.11 Disable interactive debugging – `Y`

This option applies only to the OCCAM modules only. The option directs the linker not to use library calls for channel i/o but instead use transputer instructions, resulting in faster execution. OCCAM modules cannot be interactively debugged if this option is used.

9.6 Selective linking of library modules

Library modules that are compiled for incompatible processor types or error modes are ignored by the linker. This allows library modules to be selectively loaded for specific processor types or transputer classes.

Libraries supplied with the toolset are supplied in several forms to cover the complete range of transputer types. User libraries that are likely to be used on different transputer types should be supplied for all transputer types likely to be used.

Libraries are also selected for linking on the basis of previous usage. Modules that are used by several input files are linked in only once.

9.7 The link map file

Module data and details of the target processor are always included in the linked unit output file in the form of a comment. This information may also be directed to a named output file by using the `MO` command line option.

The file contains a map of the code being linked and contains information which may assist the user during program debugging. It is generated in text format and

covers two categories of input file; separate compilation units, and library modules. The map consists of single line records containing a number of fields. Fields have a single character name followed by a colon. The following information is included:

9.7.1 MODULE record:

A module record is created for each component module in the linked unit.

Record name	Description
N	Module number assigned by the linker.
S	Source filename, may be empty if string is unobtainable.
F	Object filename, the name of the file of library from which the module has been loaded. This will be the full path name.
O	File offset, the offset (in bytes) of the module within its object file.
R	Reference, an external symbol that is used for loading the module from a library. This field will be blank if the module was not loaded from a library.
M	The compilation mode, processor type/class.

9.7.2 SECT record:

A section record is created for each section in the linked unit and shows where it is located.

Record name	Description
N	Section number assigned by the linker.
R	Name of the section.
A	Section attributes, where R – read, W – write, X – execute, D – debug, V – virtual.
P	Whether the code has been placed at a fixed address; either N (no) or Y (yes).
O	The offset in bytes of the section within the code.
S	The size in bytes of the section.

9.7.3 MAP record:

This record shows how a region of the linked unit is mapped to a module and section.

Record name	Description
M	Module number of the module that supplied this region.
R	Section number of the section in which this region lies.
A	Address of the region, in bytes.
S	Size of the region, in bytes.

9.7.4 Value record:

This record shows the value of a symbol after linkage.

Record name	Description
N	Section number assigned by the linker.
O	Name of the origin symbol – OCCAM modules only. (Used by the linker to ensure the order of compilation is correct in respect to #USE) .
M	Module number of the exporting module.
U	Whether the symbol has been used (externally at least); either N (no) or Y (yes).
V	Value of the symbol after linking. Expressed as a decimal integer or as a section number plus byte offset into that section.

9.8 Using imakef for version control

The **imakef** tool may be used to simplify the linking of complex programs, particularly those which use libraries that are nested within other libraries or compilation units.

Note: For **imakef** to function correctly the special file extension system described in section 11.3 and appendix A *must* be used.

9.9 Error messages

This section lists each error and warning message that can be generated by the linker. Messages are in the standard toolset format which is explained in appendix A.

9.9.1 Warnings

filename - bad format: reason

The named file does not conform to a recognized INMOS file format or has been corrupted.

Size bytes too large for 16 bit target

The code part of the linked unit has exceeded the address space of the T212 derived processor family.

***filename* - symbol, implementation of channel arrays has changed**

Only generated in programs where OCCAM code is used that was compiled in LFF format. The implementation of channel arrays in OCCAM differs between the earlier OCCAM 2 compiler and the current TCOFF-based configurer, and channel arrays cannot therefore be used as parameters to configured procedures.

***filename* - symbol *symbol* not found**

The specified symbol was not found in any of the supplied modules or libraries.

file1* - usage of *symbol* out of step with *file2

May be generated when linking programs incorporating OCCAM modules with a `#USE` directive, which causes the compiler to scan the file for details concerning certain program resources. This file *must* be unchanged at link time, and the message indicates that this is not the case. There are several possible causes:

- 1 *file2* has been recompiled after *file1*, in which case *file1* requires recompiling.
- 2 The file that occurred in the `#USE` directive has been replaced by a different version of the file at link time.
- 3 The file that occurred in the `#USE` directive has not been supplied to the linker, but the linker has located a different version of a required entry point elsewhere.

The OCCAM compiler `oc` may need to scan certain libraries, of which the user is unaware. Specifying one of the special OCCAM linker indirect files `occam2.lnk`, `occama.lnk` or `occam8.lnk` should take care of these 'hidden' libraries.

9.9.2 Errors

filename* - name clash with *symbol* from *filename

May be generated when linking mixed language programs incorporating OCCAM modules.

In languages such as OCCAM entry points may be scoped, i.e. extra information is associated with each symbol to indicate which version of that entry point it is. This allows programs to be safely linked even though there

are several different versions of the same entry point occurring at different lexical levels within the program.

This error indicates that a language without OCCam-type scoping has been mixed with a scoped language and a name conflict has occurred between a scoped and non scoped symbol.

filename - symbol symbol multiply defined

The symbol, introduced in the specified file, has been introduced previously, causing a conflict. The same module may have been supplied to the linker more than once or there may be two or more modules with the same entry point or data item defined.

filename - symbol symbol not found

The specified symbol was not found in any of the supplied modules or libraries.

filename - usage of symbol out of step with namefile

May be generated when linking programs incorporating modules with a **#USE** directive which causes the compiler to scan the file for details concerning certain program resources. This file *must* be unchanged at link time, and the message indicates that this is not the case. There are several possible causes:

- 1 *file2* has been recompiled after *file1*, in which case *file1* requires recompiling.
- 2 The file that occurred in the **#USE** directive has been replaced by a different version of the file at link time.
- 3 The file that occurred in the **#USE** directive has not been supplied to the linker, but the linker has located a different version of a required entry point elsewhere.

The OCCam compiler **oc** may need to scan certain libraries, of which the user is unaware. Specifying one of the special OCCam linker indirect files **occam2.lnk**, **occama.lnk** or **occam8.lnk** should take care of these 'hidden' libraries.

9.9.3 Serious errors

filename - bad format: reason

The named file does not conform to a recognized INMOS file format or has been corrupted.

filename - line number - bad format: excessively long line in indirect file

A line is too long. The length is implementation dependent, but on all currently supported hosts it is long enough so as only to be exceeded in error.

filename - line number - bad format: file name missing after directive

A directive (such as include) has no file name as an argument.

filename - line number - bad format: directive invalid number

A numeric parameter supplied to a directive does not correspond to the appropriate format.

filename - bad format: multiple main entry points encountered

A symbol may be defined to be the main entry point of a program by a compiler. Only one such symbol must exist within a single link.

filename - linenumber - bad format: non ASCII character in indirect file

The indirect file contains some non printable text. A common mistake is to specify a library or module with the 'F' command line argument or an include directive.

filename - bad format: not linkable file or library

The linker expects that all files names presented without a preceding switch (on the command line) or directive (in an indirect file) are either libraries or modules.

filename - line number - bad format: only single parameter for directive

The directive has been given too many parameters.

Cannot create output without main entry point

No main entry point has been specified.

Command line: 1k minimum for paged memory option

When using the KB option, the amount of memory used to hold the image of the program being linked is specified. There is a minimum size of 1k.

Command line: token

An illegal token has been encountered on the command line.

Command line: bad format number

A numerical parameter of the wrong format has been found.

Command line: image limit multiply specified

The command line option 'KB' has been specified more than once.

Command line: 'load and terminate' option set, some arguments invalid

Options to load and terminate the linker have been specified in conjunction with other command line options. The linker cannot execute these options if it has been instructed to terminate first.

Command line: multiple debug modes

The command line option 'Y' has been specified more than once.

Command line: multiple error modes

More than one error mode has been specified to the linker.

Command line: multiple module files specified

The command line option 'MO' has been specified more than once.

Command line: multiple output files specified

The command line option 'O' has been specified more than once.

Command line: multiple target type

More than one target processor type has been specified to the linker.

Command line: only one output format allowed

The options 'T', 'LB' and 'LC' are mutually exclusive.

***filename* - could not open for input**

The named file could not be found/opened for reading.

***filename* - could not open for output**

The named file could not be opened for writing.

***filename* - line number - could not open for reading**

The file name specified in an include directive could not be opened.

Could not open temporary file

The 'KB' option has been used in a directory where there is no write access or not enough disc space.

***filename* - mode: mode - linker mode: mode**

The linker has been given a module to link which has been compiled with attributes incompatible with the options (or lack thereof) on the linker command line.

Invalid or missing descriptor for main entry point *symbol*

Applies to occam modules only. The specified main entry point to the program does not have a valid occam descriptor. This occurs if the wrong symbol name has been used, for example a data symbol in C.

Multiple main entry points specified

The main entry point has been specified on the command line or in an indirect file more than once.

filename - line number - directive not enough arguments

The wrong number of arguments have been supplied to a directive.

filename - nothing of importance in file

The file name specified in an include directive was empty or contained nothing but white space or comments.

Nothing to link

Various options have been given to the linker but no modules or libraries.

filename - line number - only one file name per line

More than one file name has been placed on a single line within an indirect file.

filename - line number - directive too many arguments

The wrong number of arguments have been supplied to a directive.

Unknown error modes not supported in the LFF format**Unknown processors not supported in the LFF format**

When generating LFF format files, certain constructs will have no representation. For example processor types that have come into existence since the LFF format was defined.

filename - line number - unrecognised directive directive

An unrecognized directive has been found in a linker indirect file.

9.9.4 Embedded messages

Tools that create modules to be linked with **ilink** may embed "messages" within them. Three levels of severity exist; serious, warning, and message. The documentation of the appropriate tool should be consulted for more information. The format of these messages is as follows:

Serious - *ilink - filename - message: message*

Warning - *ilink - filename - message: message*

Message - *ilink - filename - message*

10 `ilist` - binary lister

This chapter describes the binary lister tool `ilist`, which takes an object file and displays information about the object code in a readable form. The chapter provides examples of display options and ends with a list of error messages which may be generated by `ilist`.

10.1 Introduction

The binary lister tool `ilist` reads an object code file, decodes it, and displays useful information about the object code on the screen. The output may be redirected to a file. Command line options control the type of data displayed.

The `ilist` tool can decode and display object files produced by `icc`, by the linker, librarian, configurer and collector tools, and by other compatible INMOS compilers such as the OCCAM 2 compiler `oc`. Files in editable ASCII format are listed without further processing.

The `ilist` tool will also list compilation and linked units in Linker File Format used by earlier versions of the INMOS toolsets (such toolsets are the IMS D705/D605/D505 and the D711/D611/D511 series toolsets).

Also, because `ilist` uses the same method to locate files as the other tools (see section A.4) it can be used to find and display the location of header files and library files in the search path specified by `ISEARCH`.

10.2 Data displays

There are several categories of data that can be displayed. Categories are selected by options on the command line. The main categories are:

- *Symbol data* – symbol names in each module. Information is displayed in tabular form.
- *External reference data* – names of external symbols used by each module. Information is displayed in tabular form.
- *Module data* – data for each module including target processor, compilation mode, and module file name.
- *Code listing* – code contained in each module, displayed in hexadecimal format.
- *Index data* – the content of library indexes.

- *Procedural data* – for external OCCam calls only.

10.2.1 Modular displays

Object code files reflect the modular structure of the original source. Single unit compilations produce a file containing a single object module, whereas units containing many compilations, such as libraries and concatenations of modules, produce object files with as many object modules. The data produced by `ilist` reflects the modular composition of object files.

10.2.2 Example displays used in this chapter

Except where indicated, the example displays used in this chapter show the output generated from the lister for the compiled (`.tco`) file generated by `icc` for the 'Hello World' example program. The program was compiled for a T425 processor.

10.3 Running the binary lister

To invoke the binary lister use the following command line:

► `ilist {filenames} {options}`

where: *filenames* is a list of one or more files to be displayed.

options is a list of one or more of the options given in Table 10.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Note: Options will only be applied to files of the appropriate file type. If the file cannot be displayed by the specified option, an error message is generated and the file is not displayed.

Example of use:

```
ilist hello.tco -a          (UNIX based toolsets)
ilist hello.tco /a         (MS-DOS and VMS based toolsets)
```

Option	Description
A	Displays all the available information on the symbols used within the specified modules.
C	Displays the code in the specified file as hexadecimal. This option also invokes the 'T' option by default.
E	Displays all exported names in the specified modules.
H	Displays the specified file(s) in hexadecimal format.
I	Displays full progress information as the lister runs.
M	Displays module data.
N	Displays information from the library index.
O filename	Specifies an output file. If more than one file is specified the last one specified is used.
P	Displays any procedural interfaces found in the specified modules.
R reference	Displays the library module(s) containing the specified reference. This option is used in conjunction with other option to display data for a specific symbol. If more than one library file is specified the last one specified is used.
T	Displays a full listing of a file in any file format.
W	Causes the lister to identify a file. The filename (including the search path if applicable) is displayed followed by the file type. This is the default option.
X	Displays all external references made by the specified modules.

Table 10.1 **ilist** command line options

ilist will attempt to identify the file type by its contents. If filenames only are supplied, **ilist** uses the default option 'w' and simply displays the file's identity.

Examples of **ilist** usage and the displays generated by the options can be found in succeeding sections.

10.3.1 Options to use for specific file types

Table 10.2 lists the available options and indicates which file formats they may be used to list. The table also lists the file types it is recommended to use with each option, in order of usefulness.

Option	Permitted file format	Recommended usage
H	Any format	
O	Any format	
T	Any format	
W	Any format	
A	TCOFF only	.lib, .tco, .lku
C	TCOFF only	.tco, .lku, .lib
E	TCOFF only	.lib, .tco, .lku
M	TCOFF only	.tco, .lku, .lib
N	TCOFF libraries only	.lib
P	TCOFF only	.lib, .tco, .lku
R	TCOFF libraries only	.lib
X	TCOFF only	.lib, .tco, .lku

Table 10.2 Recommended options

10.3.2 Output device

ilist sends its output to the host standard output stream, normally the terminal screen. Facilities available on the host system may allow you to redirect the output to a file, or send it to another process, such as a sort program. For details of these facilities consult the documentation for your system. Alternatively the **ilist** 'O' command line option may be used to redirect output to a specified file.

10.3.3 Default command line

A set of default command line options can be defined for the tool using the **ILISTARG** environment variable. Options must be specified using the syntax required by the command line.

10.4 Specifying an output file – option o

The **O** option enables the user to redirect the display data to an output file. If more than one output file is specified on the command line then the last one specified is used. File extensions should be specified, because defaults are not assumed.

Display options are described in the following sections 10.5 to 10.15. Options are given in alphabetical order.

10.5 Symbol data – option **a**

This option displays all the available information about the symbols used within the specified modules. A tabular format is used.

Note: The data produced by this display is extensive and detailed and assume some knowledge of the object file format.

The following information is given:

- Symbol name.
- Section attributes, if applicable.
- Symbol attributes.
- The number of the symbol within the module plus the number of its origin.
- Module name.
- Target processor.
- Error mode.
- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

10.5.1 Specific section attributes

Certain attributes apply only to symbols which are section names. If they are applicable, these attributes are indicated by the following nomenclature and displayed as a character string:

R	– Read section.
W	– Write section.
X	– Execute section.
D	– Debug section.
V	– Virtual section.

10.5.2 General symbol attributes

Attributes for all symbols, including section names, are also indicated by a character string, using the following nomenclature:

Symbol	Description attribute
L	Symbol local to the module.
E	Symbol exported from the module.
I	Symbol imported to the module.
W	Weak attribute, indicates that the symbol takes the value 0 when not defined.
C	Conditional attribute, indicates that the first value given to the symbol is always used.
U	Unindexed, indicates that the symbol is not present in the library index.
P	Provisional attribute, indicates that the last value given to the symbol is always used.
O	Indicates that the symbol is an origin symbol. The origin symbol is used by the linker to check the origin of the module.

Symbol attributes are displayed immediately after the section attributes, and each attribute is displayed at a specific position in the string. Attributes which are not present are indicated by a hyphen '-'.

The position of each attribute in the string is as follows:

RWXDV LEIWCUPO

10.5.3 Example symbol data display

Figure 10.1 shows the symbol data display for the compiled file **hello.tco**.

module%table%base	----V -E-----	0	hello.c	T425 X
module%number	----- L-----	1	hello.c	T425 X
static%base	----V -E-----	2	hello.c	T425 X
local%static	----- L-----	3	hello.c	T425 X
%lsb	----- L-----	4	hello.c	T425 X
text%base	R-X-- -E-----	5	hello.c	T425 X
local%text	----- L-----	6	hello.c	T425 X
next%common	----- -E--CU--	7	hello.c	T425 X
main	----- -E-----	8	hello.c	T425 X
_IMS_printf	----- -I-----	9	hello.c	T425 X
static%space	----- -E--UP-	10	hello.c	T425 X

Figure 10.1 Example output produced by the **A** option.

10.6 Code listing – option C

The 'C' option produces a full listing of the code in the same format as that generated by the 'T' option, but with the addition of a hex listing of the code at each **LOAD TEXT** directive. This option *may* be accompanied by the 'T' option; if the 'T' option is not specified it is supplied automatically.

The output from this option gives an ASCII dump, in hexadecimal format, of the code for each module. It can be used on any object code.

When used to display object code produced by the occam compiler, the code for each module is displayed as a contiguous block of lines, where each line has the format:

address *ASCII hex* *ASCII characters*

where: *address* is the address of the first byte on the line, expressed as an offset from the start of the module.

ASCII hex is the hex representation of the code

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable it is replaced by a dot '.'.

10.6.1 Example code listing display

Figure 10.2 shows the code listing display for the compiled file `hello.tco`.

```
00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP DUP WSUBDB MOVE2D CRC BITOPS FPTSTERR
LDDEVID DBGSUP TMRDIS LDMSTVL POP BIT32 MS=28 ICALL X lang: ANSI_C ""
00000010 VERSION tool: icc origin: hello.c
0000001E SECTION VIR EXP "module%table%base" id: 0
00000034 SET_LOAD_POINT id: 0
00000037 SYMBOL LOC "module%number" id: 1
00000048 DEFINE_LABEL id: 1
0000004B SECTION VIR EXP "static%base" id: 2
0000005B SET_LOAD_POINT id: 2
0000005E SYMBOL LOC "local%static" id: 3
0000006E DEFINE_LABEL id: 3
00000071 SYMBOL_LOC "%lsb" id: 4
00000079 DEFINE_SYMBOL id: 4 SS:0+SV:3
00000081 SECTION REA EXE EXP "text%base" id: 5
0000008F SET_LOAD_POINT id: 5
00000092 SYMBOL_LOC "local%text" id: 6
000000A0 DEFINE_LABEL id: 6
000000A3 SYMBOL EXP CON UNI "next%common" id: 7
000000B2 DEFINE_SYMBOL id: 7 SS:2
000000B7 COMMENT bytes: 5
000000C1 LOAD_EXPR size: 4 SV:1
000000C6 SYMBOL EXP "main" id: 8
000000CE DEFINE_SYMBOL id: 8 SV:6+4
000000D6 SYMBOL IMP "_IMS_printf" id: 9
000000E5 LOAD_TEXT bytes: 24
000000E8 4521FB71 219222F0 0A48656C 6C6F2057 E!..q!.."..Hello W
000000F8 6F726C64 0A002020 orld..
00000100 LOAD_PREFIX size: 6 AP(SV:9-LP) instr: j
00000109 LOAD_TEXT bytes: 2
0000010C 2020
0000010E COMMENT bytes: 33
00000134 SYMBOL EXP UNI PRO "static%space" id: 10
00000144 DEFINE_SYMBOL id: 10 SV:7
00000149 END_MODULE
```

Figure 10.2 Example output produced by the C option

10.7 Exported names – option **E**

The output from this option is in a tabular format. It consists of a list of names exported by the modules. This option also displays any globally visible data.

The following information is given by the display:

- Exported name.
- The name of the module in which the exported name is found.
- Language used.
- Target processor.
- Error mode.
- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

10.7.1 Example exported names display

Figure 10.3 shows the exported names display for the compiled file `hello.tco`.

main	-> hello.c	ANSI_C	T425 X
------	------------	--------	--------

Figure 10.3 Example output produced by the **E** option

10.8 Hexadecimal/ASCII dump – option **x**

This option provides a display of the specified files in hexadecimal and ASCII format. The option does not attempt to identify file types and may be used to display any files which the lister has previously identified incorrectly.

The output takes the form of a hexadecimal representation of the whole of the file content. The display has a similar appearance to that produced by the **C** option, however, the **C** option only functions on *code* found within the file.

Each module is displayed as a contiguous block of lines, where each line has the format:

address *ASCII hex* *ASCII characters*

where: *address* is the address of the first byte on the line, expressed as an offset from the start of the module.

ASCII hex is the hex representation of the code

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable it is replaced by a dot '.'.

10.8.1 Example hex dump display

Figure 10.4 shows the hex dump display for the compiled file `hello.tco`.

```

00000000 0100020C FDFFE1F 00FD52E0 07000400 .....R....
00000010 1B0C0369 63630768 656C6C6F 2E630B14 ...icc.hello.c..
00000020 1002116D 6F64756C 65257461 626C6525 ...module$table%
00000030 62617365 0401001E 0F010D6D 6F64756C base.....modul
00000040 65256E75 6D626572 0E01010B 0E10020B e%number.....
00000050 73746174 69632562 61736504 01021E0E static%base.....
00000060 010C6C6F 63616C25 73746174 69630E01 ..local%static..
00000070 031E0601 04256C73 620F0604 06040003 .....%lsb.....
00000080 030B0C06 02097465 78742562 61736504 .....text%base.
00000090 01051E0C 010A6C6F 63616C25 74657874 .....local%text
000000A0 0E01061E 0D320B6E 65787425 636F6D6D .....2.next%comm
000000B0 6F6E0F03 07040214 08000005 0D008194 on.....
000000C0 06080304 03011E06 02046D61 696E0F06 .....main..
000000D0 08060306 01041E0D 040B5F49 4D535F70 ....._IMS_p
000000E0 72696E74 66061918 4521FB71 219222F0 printf...E!..q!..
000000F0 0A48656C 6C6F2057 6F726C64 0A002020 .Hello World..
00000100 0707060D 07030902 00060302 20201424 ..... $.
00000110 00002103 03010001 0E050404 03060800 ..!.....
00000120 000E0908 040E0B0A 040E0D0C 0A040E04 .....
00000130 0F0C101C 1E0E620C 73746174 69632573 .....b.static%
00000140 70616365 0F030A03 07030000 pace.....

```

Figure 10.4 Example output produced by the `H` option

10.9 Module data – option `M`

This option displays any header information which is present. This may include version control data, general comments that may have been appended to the file during use of the toolset and copyright information. The data is displayed for individual modules in the object file and includes:

- Module name
- Transputer type and compilation error mode
- Language type
- Version control data
- Comments inserted by the toolset, for example, copyright clauses.

Data is displayed in separate blocks for each module. Some of the data is also used by other tools in the toolset, for example, some comments are used by the debug-

ger tool `idebug` while version information is used by some tools for compatibility testing.

When *linked* units are displayed using this option, a long comment will be displayed. This comment gives details of the allocation of memory to each separately compiled code and library module used in the linked module. The following information is given in tabular format:

- Code type - Separately compiled code (SC) or library module (LIB).
- Module name.
- Address offset in linked module.
- Start address.
- End address.
- Reference in library (if applicable) used to locate the relevant library module.

10.9.1 Example module data display

Figure 10.5 shows the module data display for the compiled file `hello.tco`.

```
MODULE: ANSI_C      T425 X  
VERSION: icc hello.c
```

Figure 10.5 Example output produced by the **M** option

10.10 Library index data – option **N**

This option is used to list library indexes. The data is given in a tabular format. For each entry in the index the following information is given:

- The address of the module in the library.
- The symbol name.
- The language the module is written in.
- The target processor type.
- The error mode used.

- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

10.10.1 Example library index display

Figure 10.6 shows part of the output produced by the 'N' option for one of the standard C library files.

```
00025C21 ie64op.pax:8340AC71      OCCAM      TA  X
00036155 xlink1.pax:F11BAD5A      OCCAM      TA  X
00034AF7 DATAN2%c                 OCCAM      TA  X
000330D0 DCOS%c                   OCCAM      TA  X
0000B898 DefaultSignalHandler%c   ANSI_C     TA  X
0001CAC6 floorf                   ANSI_C     TA  X
0002007B get_static_size%c        ANSI_C     TA  X
000129AD sub_vfprintf%c           ANSI_C     TA  X
```

Figure 10.6 Example output produced by the N option

10.11 Procedural interface data – option p

This option is only applicable to occam modules or mixed language programs. It displays procedural interface information for all external occam functions and procedures. The following information is displayed for each module:

- Target processor.
- Error mode.
- Language used.
- Amount of workspace used by the procedure or function, in words.
- Amount of vector space used by the procedure or function, in words.
- Parameters used by the procedure or function.
- Data type of parameters.
- Channel usage, if applicable.

Channel usage is displayed in occam notation. A channel marked with an ? is an *input* channel to the code of that entry point, and a channel marked with ! is an *output* channel.

When a library file is listed this will be indicated by the words 'INDEX ENTRY mode:' rather than 'DESCRIPTOR mode'.

10.11.1 Example procedural data display

Figure 10.7 shows an example procedural data display for a compiled occam module. This example is taken from the 'simple' example occam program compiled by oc for the TA processor class.

```

DESCRIPTOR mode: TA  H   language: OCCAM      <ORIGIN DESCRIPTOR>
DESCRIPTOR mode: TA  H   language: OCCAM
ws: 75 vs: 128
PROC simple(CHAN OF SP fs,
CHAN OF SP ts,
[ ]INT memory)
  SEQ
    fs?
    ts!
:

```

Figure 10.7 Example output produced by the P option

10.12 Specify reference – option R

This option is used in conjunction with any of the other display options to locate a specific symbol within a named library. All library modules that export the symbol are displayed.

The exact format of the display depends on the main display option with which R is used.

Note: Symbol names must be specified in the correct case.

10.13 Full listing – option T

This option displays all *data* found in the input file. Provided that *ilist* recognizes the file type, the file is decoded in its own format. Text file are displayed as text and unrecognized file types are displayed as a hexadecimal dump.

Data is not displayed in a tabular form but is output in the sequence in which it is found in the module.

The display formats are tailored to each file format and are intended for diagnostic support and analysis; large amounts of data are produced which may require skilled interpretation.

10.13.1 Example full data display

Figure 10.8 shows the full data display for the compiled file `hello.tco`.

```

00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP DUP WSUBDB MOVE2D CRC BITOPS FPTSTERR
  LDDEVID DBGSUP TMRDIS LDMSTVL POP BIT32 MS=28 ICALL X lang: ANSI_C ""
00000010 VERSION tool: icc origin: hello.c
0000001E SECTION VIR EXP "module%table%base" id: 0
00000034 SET_LOAD_POINT id: 0
00000037 SYMBOL LOC "module%number" id: 1
00000048 DEFINE_LABEL id: 1
0000004B SECTION VIR EXP "static%base" id: 2
0000005B SET_LOAD_POINT id: 2
0000005E SYMBOL LOC "local%static" id: 3
0000006E DEFINE_LABEL id: 3
00000071 SYMBOL LOC "%lsb" id: 4
00000079 DEFINE_SYMBOL id: 4 SS:0+SV:3
00000081 SECTION REA EXE EXP "text%base" id: 5
0000008F SET_LOAD_POINT id: 5
00000092 SYMBOL LOC "local%text" id: 6
000000A0 DEFINE_LABEL id: 6
000000A3 SYMBOL_EXP CON UNI "next%common" id: 7
000000B2 DEFINE_SYMBOL id: 7 SS:2
000000B7 COMMENT bytes: 5
000000C1 LOAD_EXPR size: 4 SV:1
000000C6 SYMBOL_EXP "main" id: 8
000000CE DEFINE_SYMBOL id: 8 SV:6+4
000000D6 SYMBOL_IMP "_IMS_printf" id: 9
000000E5 LOAD_TEXT bytes: 24
00000100 LOAD_PREFIX size: 6 AP(SV:9-LP) instr: j
00000109 LOAD_TEXT bytes: 2
0000010E COMMENT bytes: 33
00000134 SYMBOL_EXP UNI PRO "static%space" id: 10
00000144 DEFINE_SYMBOL id: 10 SV:7
00000149 END_MODULE

```

Figure 10.8 Example output produced by the T option

10.13.2 Configuration data files

The full data listing of a configured (.cfb) file shows how the processes are mapped onto a transputer system and has a different appearance to other displays produced by this option.

10.14 File identification – option w

This option causes the lister to identify the file type. `ilist` takes a heuristic approach to file identification. The filename is displayed along with the file type. The full path to the file is also displayed if the file is not in the current directory (i.e. if it has been found in the search path specified in the `ISEARCH` environment variable). This is the default command line invocation if no other option is supplied.

Table 10.3 indicates how the lister classifies file types.

File format	Default extension	Listed file type
TCOFF compiled unit	.tco	TCOFF LINKABLE UNIT
TCOFF compiled library unit	.lib	TCOFF LINKABLE UNIT LIBRARY
TCOFF linked unit	.lku	TCOFF LINKED UNIT
TCOFF linked library unit	.lib	TCOFF LINKED UNIT LIBRARY
Configuration binary	.cfb	CONFIGURATION BINARY
Core dump	.dmp	CORE DUMP FILE
Network dump	.dmp	NETWORK DUMP
LFF file	.cxx, .txx	LFF SC
LFF library	.lib	LFF LIBRARY
Extracted SC	.rxx	EXTRACTED SC
iboot program	.bxx	BOOTABLE PROGRAM (iboot)
Extracted program	.bt1	BOOTABLE PROGRAM
Empty file	–	EMPTY FILE
Text files	–	TEXT FILE
None of the above	–	UNKNOWN BINARY FORMAT

Table 10.3 File types recognised by `ilist`

where: SC files are separately compiled files.

LFF files are separately compiled or linked files in LFF format.

Extracted files are files which have been compiled and developed to be dynamically loaded onto a transputer system.

iboot programs are programs which have had a bootstrap added by the iboot tool, supported by previous issues of the toolset i.e. the IMS D711/D611/D511 and D705/D605/D505 series toolsets.

10.14.1 Example file identification display

Figure 10.9 shows the file identification display for the compiled file `hello.tco` and two linker control files. This output was generated by the following command:

```
ilist hello.tco occama.lnk clibsrld.lnk
```

hello.tco	TCOFF LINKABLE UNIT
/home/D4205/libs/occama.lnk	TEXT FILE
/home/D4300/libs/clibsrld.lnk	TEXT FILE

Figure 10.9 Example output produced by the **w** option

10.15 External reference data – option **x**

This option displays a list of all the code and data symbols imported by the modules specified to the lister, i.e. it lists their external references. External references are references to separately compiled units. For C programs the option will also display any external references to globally visible data.

The output from this option is in a tabular format. It consists of a list of external references and their associated modules. The following information is displayed:

- External reference i.e. name of the separately compiled unit.
- The name of the module in which the external reference exists.
- Language used.
- Target processor.
- Error mode.
- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

10.15.1 Example external reference data display

Figure 10.10 shows the external reference data display for the compiled file `hello.tco`.

<code>_IMS_printf</code>	<code><- hello.c</code>	<code>ANSI_C</code>	<code>T425 X</code>
--------------------------	----------------------------	---------------------	---------------------

Figure 10.10 Example output produced by the **x** option

10.16 Error messages

This section lists error and warning messages that can be generated by the lister. Messages are in the standard toolset format which is explained in appendix A.

10.16.1 Warning messages

filename - reason

The named file does not conform to a recognized INMOS file format or has been corrupted.

10.16.2 Serious errors

filename - bad format: reason

The named file does not conform to a recognized INMOS file format or has been corrupted.

filename - could not open for input

The named file could not be found/opened for reading.

filename - could not open for output

The named file could not be opened for writing.

filename - file type does not correspond to command line options

The options given to the lister apply to formats dissimilar to the format of the file being read.

must supply additional TCOFF options with reference *reference*

The required format of the listing has not been specified.

*filename - no entry for *reference* in library index*

The specified reference cannot be found in the library index.

parsing command line *token*

An unrecognized token was found on the command line.

filename - unexpected end of file

The named file does not conform to a known INMOS file format or has been corrupted.

11 **imakef** — makefile generator

This chapter describes the makefile generator **imakef** that creates makefiles for input to make programs. It explains how the tool can be used to create makefiles and describes the special file naming conventions that allow **imakef** to create makefiles for mixtures of code types. The chapter describes the format of makefiles generated by **imakef** and ends with a list of error messages.

11.1 Introduction

Make programs automate program building by recompiling only those components that have been changed since the last compilation. To do this they read a **makefile** which contains information about the interdependencies of files with one another, along with command lines for rebuilding the program.

imakef creates makefiles for all types of toolset object files, using its built in knowledge of how files referenced within the target file depend on one another. It is intended to be used with all INMOS compiler systems that generate TCOFF object code, which includes the the ANSI C compiler **icc**, the OCCAM 2 compiler **oc** and the FORTRAN 77 compiler **if77**. Its mode of operation with different languages is controlled by command line options. The makefile is generated in a standard format for input to most make programs.

Makefiles created using **imakef** are compatible with many public domain and proprietary make programs. The following make programs are directly compatible:

- Borland **make**.
- Unix **make**.
- Microsoft **nmake**.
- Gnu **make**.

However, the older Microsoft make program "**make**" is *not* compatible.

The source of **imakef** is supplied with the toolset so that it can be modified for use with other make programs.

11.2 How **imakef** works

imakef operates by working back from the target file to determine its dependencies on other files, using its knowledge of inputs and outputs of each tool and

the compilation architecture of the toolset. For example, compiled object files must be created from language source files using the compiler.

In a similar way linked files must be generated from compiled files. **imakef** assumes that programs targeted at a single transputer are not required to be configured. Bootable files may therefore be generated from linked units or configuration data files. **imakef** works back from the target file, determining file dependencies and creating commands to recreate the target file, recompiling and relinking where necessary.

11.3 File extensions for use with **imakef**

imakef identifies files and file types by a special set of file extensions which identify the transputer target type and compilation error mode. This allows the tool to produce makefiles for mixed module combinations.

Note: The extensions that **imakef** requires differ in most cases from the standard toolset default extensions which are described in section A.5. For **imakef** to work correctly the extensions described in section 11.3.1 must be used on all intermediate and target files, at all stages of program development i.e. compiling, linking, configuring, and booting.

The file naming convention uses a three-character extension which identifies the type of file and in most cases includes the transputer target and error mode. Source files for the most part use standard language extensions.

11.3.1 Target files

The following table lists the types of object code files for which **imakef** can create makefiles, along with the file extension formats that must be used.

Target file	File extension
Compiled code.	.tXX
Linked code.	.cXX
Bootable code for single transputer programs.	.bXX
Bootable code for multitransputer programs.	.bt1
Dynamically loadable code.	.rXX
Libraries.	.lib
Configuration binary file.	.cfb
Library usage file.	.liu
Library indirect file.	.lbb

Compiled, linked, bootable and non-bootable files, whatever their language origin, have a transputer target designator as the *second* character of the extension, and

an error mode designator as the *third* character. Accepted values of these designators are listed below.

2nd Character	Transputer types supported
2	T212, T222, M212
3	T225
4	T414
5	T426, T425, T400
8	T800
9	T805, T801
a	Class TA
b	Class TB

3rd Character	Error mode
x	UNIVERSAL
h	HALT
s	STOP

Examples:

.t4x – refers to a compiled module targetted for T4 transputers, in UNIVERSAL error mode.

.tah – refers to a module targetted for the any 32-bit transputer in HALT error mode.

Compiled code generated by `icc` or `if77` is in UNIVERSAL mode, designated by the character 'x'. HALT and STOP code can be generated by the occam 2 compiler `oc`.

Transputer types are explained further in section B.2.

Program development using **imakef** and the extensions to use are illustrated in Figure 11.1. Target files which can be created by **imakef** are shown in bold.

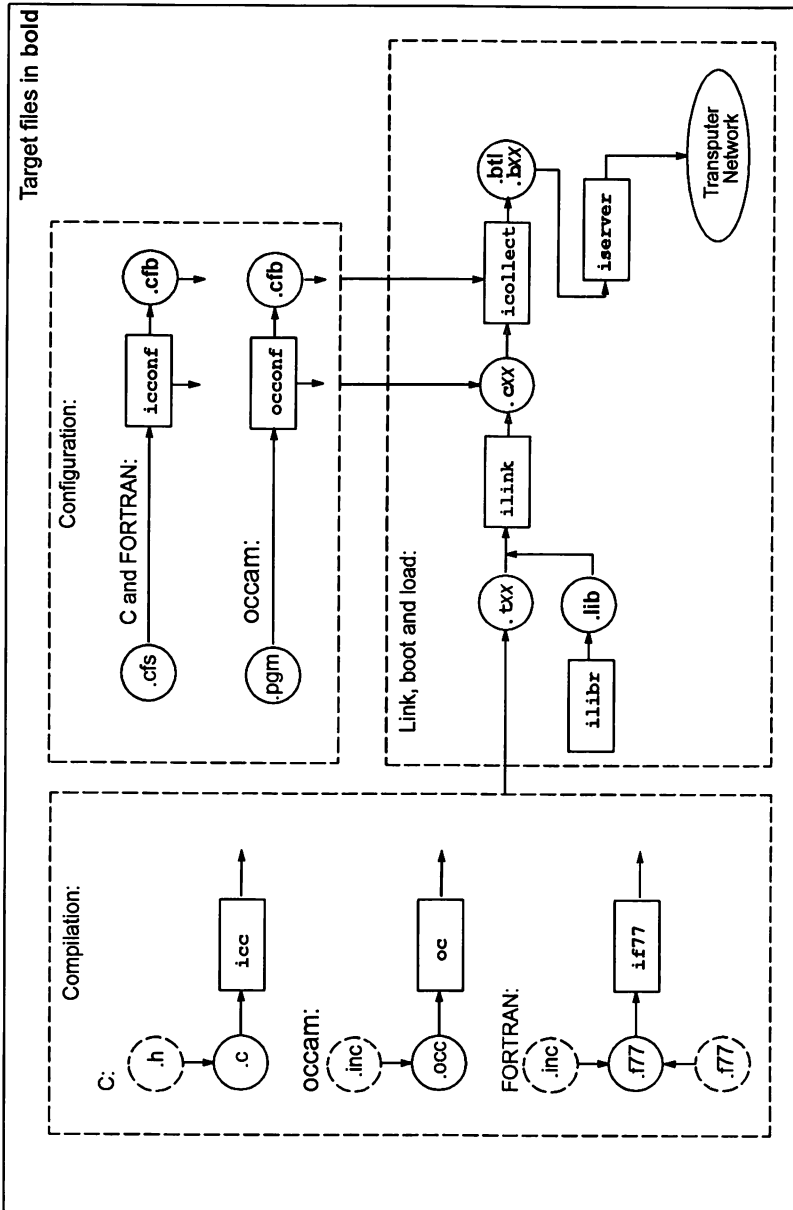


Figure 11.1 Main target files showing extensions required

11.4 Linker indirect files

For C and FORTRAN modules linker indirect files must be created for all linked units where **imakef** will be used to generate a target file. Linker indirect files define to **imakef** the components of the linked unit, providing a starting point for working out file dependencies.

Linker indirect files must be named after the linked unit to which they relate and carry the **.lnk** extension.

For programs written wholly in OCCam, **imakef** will automatically generate a linker indirect file. The file is named after the target filename but is given an extension in the form **.lxx**. The file contains a list of modules to be linked. In addition an **#INCLUDE** statement references a further linker indirect file, referencing compiler libraries. **imakef** deduces the compiler libraries to be included from the extension of the linked object file.

Section 11.6.2 provides a short description of linker indirect files and several examples are given in section 11.7.

11.5 Library indirect and library usage files

When building a library using **imakef**, a file must be provided that contains the names of all the object modules required to build the library. This file is known as a library indirect file and has the extension **.lbb**. See chapter 8 for further details.

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They contain a list of files to which the library must be linked before it can be run, and ensure that the correct linker commands are generated.

Library usage files should be created for all user-defined libraries where the source of the library is not available. They are created using **imakef**.

Library usage files are given the same name as the library to which they relate, but with a **.liu** extension. To create a library usage file using **imakef**, specify the library name and add a **.liu** extension. For example, the following command creates a library usage file for the library **mylib.lib**:

```
imakef mylib.liu
```

When **imakef** is used to create a library usage file no makefile is generated.

11.6 Running the makefile generator

The **imakef** tool takes as input a list of files generated by tools in the toolset and generates a makefile, containing full instructions of how to build the application program. The output file is named after the first target filename and is given a **.mak** extension (if no output file is specified on the command line).

To invoke **imakef** use the following command line:

► **imakef** *filenames* { *options* }

where: *filenames* is a list of target files for which makefiles are to be generated. If more than one file is specified the single makefile generated will generate all of the specified files.

options is a list, in any order, of one or more options from Table 11.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be given in any order.

Options may be entered in upper or lower case and can be given in any order.

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
C	This option is used when incorporating C or FORTRAN modules into the program. It specifies that the list of files to be linked is to be read from a linker indirect file. This option <i>must</i> be specified for correct C or FORTRAN operation.
D	Disables the generation of debugging information in compilations. The default is to compile with full debugging information.
I	Displays full progress information as the tool runs.
M	Produce compiler, linker and collector map files for imap .
NI	Files in the directories in ISEARCH are not put into the makefile. This means that system files are not present, making it much easier to read.
O filename	Specifies an output file. If no file is specified the output file is named after the target file and given the .mak extension.
R	Writes a deletion rule into the makefile.
Y	Disables interactive (breakpoint) debugging in all compilations. The default is to compile with full breakpoint debugging information.

Table 11.1 **imakef** options

11.6.1 Example of use

imakef hello.b4x -c (UNIX based toolsets)

imakef hello.b4x /c (MS-DOS and VMS based toolsets)

This creates the makefile **hello.mak** which when used as input to make generates the bootable file **hello.b4x** (a bootable file for T4 transputers).

11.6.2 Specifying language mode

imakef can be used with all compilers in the INMOS TCOFF family. This includes the ANSI C compiler **icc**, the OCCAM compiler **oc** as well as the FORTRAN compiler **if77**.

imakef has two modes of operation: one for the traditional languages FORTRAN and C, and another for OCCAM. OCCAM mode is the default (for historical reasons only – it was the first transputer toolset to be developed); FORTRAN and C operation is controlled by a command line option.

In OCCAM programs, file dependencies are wholly deducible from the source and target files. In FORTRAN and C programs the list of files to be processed by the linker must be created in a linker indirect file; the **imakef** 'C' option is then used to specify the linker indirect file(s) to be read. The linker indirect files must include all the components of a program, including any libraries that are used.

The 'C' option must be specified for all C and FORTRAN programs and for any mixed language programs which incorporate modules in these languages. For mixed language programs all files which are to be linked must be listed in the linker indirect file, including any OCCAM modules or library files. In systems that use mixtures of code compiled for different transputer types and error modes, a separate linker indirect file must be created for each.

An example is given in section 11.7.3 of how **imakef** may be used to build a mixed language program.

11.6.3 Configuration description files

When **imakef** builds a makefile for a configured program it will look for the presence of a configuration description file which has the same name as the program to be built.

The type of file searched for depends on the mode of operation specified to **imakef**. If the default OCCAM mode is used, that is, the 'C' option is not specified, **imakef** will look first for a configuration description file with the extension **.pgm**, (readable by the OCCAM configurer **occonf**). If a **.pgm** file is not present **imakef** will then look for a **.cfs** file (readable by the 'C-style' configurer **icconf**).

If the FORTRAN/C mode is used, that is, the 'C' option is specified, the reverse sequence is used, that is, **imakef** looks first for a **.cfs** file.

11.6.4 Disabling debug data

Two options to **imakef** disable the creation of debug data.

The 'D' option disables the generation of all debugging information in the target file. If this option is used the resulting target code cannot be debugged.

The 'Y' option disables only the data required for interactive (breakpoint controlled) debugging. If this option is given no breakpoint debugging operations can be used on the final program. Post-mortem debugging is unaffected.

11.6.5 Removing intermediate files

Intermediate files can be removed by specifying the 'R' option to **imakef**. This adds a *delete* rule to the makefile which directs make to remove all intermediate after the program is built. The delete operation is only honored if make is subsequently invoked with the DELETE option.

11.6.6 Files found on ISEARCH

When **imakef** runs, it includes all dependencies in the set of rules. The NI option prevents **imakef** recording in the makefile, any dependencies on files found using ISEARCH. As a result the makefile is easier to read and is more portable.

11.6.7 Map file output for imap

Using the 'M' option, **imakef** can be made to generate switches in the calls to the compiler, linker or collector to output map files. These map files are then available for reading by the **imap** tool, details of which can be found in chapter 12.

11.7 **imakef** examples

This section contains several examples of the use of **imakef** with different programming languages. The final example shows how a mixed language program can be built with **imakef**.

The sources appropriate to the toolset are supplied in the **imakef** examples subdirectory.

11.7.1 C examples

The first example shows how to create a makefile for a multi-module program, written in C, running on a single transputer. The second example shows how to create a makefile for a configured C program.

Single transputer program

This first example is for a program which is not configured.

The example program is made up of three source files, written in C:

```
main.c
hellof.c
worldf.c
```

imakef needs to know the names of the main components of the program, and looks for the associated linker indirect file **hello.lnk**:

hello.lnk must contain the following text:

```
main.t4x
hellof.t4x
worldf.t4x
#include cnonconf.lnk
```

Note: the use of the **.t4x** extension rather than **.tco**. This is because **imakef** needs to work out the required processor type. The startup linker indirect file **cnonconf.lnk** is also included. The inclusion of this file is standard for all C programs which are not configured and directs **imakef** to include the libraries. To create the makefile use the command:

```
imakef hello.b4x -c                                (UNIX based toolsets)
imakef hello.b4x /c                                (MS-DOS and VMS based toolsets)
```

Note: the use of the **.b4x** extension instead of **.bt1**. Using this form of extension informs **imakef** that we wish to create a bootable program for a single transputer without the aid of the configurer. The makefile **hello.mak** is created.

Multitransputer program

This example program uses the configurer to place linked units on two processors. The program is made up of the following source files written in C:

```
master.c
mult.c
multi.cfs
```

The `.cfs` file is the configuration description file. It places 2 linked units on 2 processors, using the following statements:

```
use "master.c8x" for master;
use "mult.c4x" for mult;
```

Note: the use of the `.cxx` form of extension instead of the toolset default extension for linked units `.lku`. **imakef** reads the `.cfs` file and determines that the program is made up of two linked units, each of which must have an associated linker indirect file, namely, `master.lnk`, and `mult.lnk`.

The two linker indirect files must contain the following text:

master.lnk:	mult.lnk:
<code>master.t8x</code>	<code>mult.t4x</code>
<code>#include cstartup.lnk</code>	<code>#include cstartrd.lnk</code>

Again note the use of the `.txx` form of extension. `master.lnk` includes the linker indirect file `cstartup.lnk`, which is used for configured programs linked with the full runtime library. `mult.lnk` includes `cstartrd.lnk`, the standard linker indirect file used for configured programs linked with the reduced library. This library can be used by `mult.t4x` because the module does not require host access.

To create the makefile use the following command:

```
imakef multi.btl -c (UNIX based toolsets)
imakef multi.btl /c (MS-DOS and VMS based toolsets)
```

The `.btl` extension informs **imakef** that the target is a configured program, to be built from a configuration description file called `multi.cfs`. The makefile `multi.mak` is created.

Note: when `multi.mak` invokes the configurer, the following warning is issued:

Warning-icconf-Using single hop software virtual links

This warning may be ignored.

11.7.2 occam examples

Two examples are again provided, the first for a multi-module program running on a single transputer and the second example for a configured program.

The program used is the pipeline sorter program which is supplied in the examples directory.

Single transputer program

The example program is made up of four source files, written in occam:

```
sorthdr.inc
element.occ
inout.occ
sorter.occ
```

To create the makefile use the following command:

```
imakef sorter.b4h
```

Note the use of the **.b4h** extension instead of **.bt1**. Using this form of extension informs **imakef** that we wish to create a bootable program for a single transputer without the aid of the configurer.

The makefile generator has built-in knowledge of the file name rules for occam. In this example, it knows by examining the file name that the program to be built is for a single T414 processor in HALT mode, and that the source of the main body of the program is in the file **sorter.occ**. It reads the file **sorter.occ** and discovers that it uses a library called **hostio.lib**, the two compilation units **inout** and **element**, and two include files, **sorthdr.inc** and **hostio.inc**. It then reads the sources of the include files and compilation units and finds no more file dependencies.

With this information about source file and their dependencies, **imakef** builds a makefile called **sorter.mak** containing full instructions on how to build the program and creates a linker indirect file **sorter.l4h** (see section 11.4).

To build the program run the make program on **sorter.mak**. The entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

Multitransputer program

This version of the sorter program is configured to place linked units on four processors. The program is made up of the following Occam files:

```
sorthdr.inc
element.occ
inout.occ
sortb3c.pgm
```

To create the makefile use the following command:

```
imakef sortb3c.btl
```

The `.btl` extension informs **imakef** that the target is a configured program, to be built from a configuration description file called `sortb3c.pgm`. The configuration description references two linked units:

```
#USE "inout.c4h"
#USE "element.c4h"
```

Note: the use of the `.cxx` form of extension instead of the toolset default extension for linked units `.lku`. **imakef** reads the `.pgm` file and will produce a file called `sortb3c.mak` containing a make description of the program.

To build the program run the make program on `sortb3c.mak`.

11.7.3 Mixed language program

This example, uses a mixed language program which combines both Occam and C modules. It is based on the example given in the 'Mixed language programming' chapter of the accompanying *Toolset User Guide*.

The example program is made up of the following files:

<code>mixed.t4h</code>	– Compiled occam module
<code>cfunc.t4x</code>	– Compiled C module

where: the occam module is the main program which calls in the C function.

To create the makefile for the example program use one of the following commands:

<code>imakef callc.b4h -c</code>	(UNIX based toolsets)
<code>imakef callc.b4h /c</code>	(MS-DOS and VMS based toolsets)

This command informs `imakef` that we wish to create a bootable program for a single T414 processor in HALT mode. The 'c' option tells `imakef` that the program also includes modules written in C.

`imakef` needs to know the names of the C components of the program, and looks for the associated linker indirect file `callc.lnk`. Because a linker indirect file is supplied to `imakef`, all the modules to be linked must be listed.

`callc.lnk` must contain the following files:

```
callc.t4x
csubfunc.t4h
callc.lib
hostio.lib
#include clibsrcd.lnk
#include occama.lnk
```

The Occam module is listed first, because it contains the main entry point of the program. **Note:** the use of the `.t4h` and `.t4x` extensions. The C module has been compiled in UNIVERSAL mode, which is the standard mode for the C compiler. This does not cause a problem because UNIVERSAL mode may be called by HALT mode.

The files `hostio.lib` and `callc.lib` are the Occam libraries. `occama.lnk` contains a list of Occam compiler libraries which may be required.

`clibsrcd.lnk` references the reduced C runtime library used by the C module. **Note:** `clibsrcd.lnk` does not contain an entry point. When building a C program which calls in other language modules, a C main entry point is required. Therefore

one of the standard C startup files should be used i.e. `cstartup.lnk`, `cstartrd.lnk` or `cnonconf.lnk`.

With this information `imakef` builds the make program `callc.mak`.

Further information about mixed language programming can be found the accompanying *Toolset User Guide*.

11.8 Format of makefiles

Makefiles essentially consist of a number of *rules* for building all the parts of a program. Each rule contains two main elements: a definition of the file's dependencies in a format acceptable to make programs; and the command to recreate the file on a specific host. All makefiles also contain macros which define command strings and option combinations.

11.8.1 Macros

All makefiles created by `imakef` include a set of macro definitions inserted at the head of the file.

Macros define strings which are used to call the compiler, the configurer, the linker, the librarian, the collector, and the eprom formatter tools, and fixed combinations of options for these tools.

Macros are provided so that customized versions of the toolset commands, and specific combinations of options, can be easily incorporated. Existing macros can be modified for specific host environments, and new macros created, by editing the makefile.

The full set of macros defined by `imakef` can be found by consulting any makefile created by the tool.

11.8.2 Rules

Rules define the dependencies of object files on other files and specify *action strings* to build those files.

Example:

UNIX based toolsets:

```
example.t4h : example.c
$(CC) example -t4 -h -o example.t4h $(COPT)
```

MS-DOS and VMS based toolsets:

```
example.t4h : example.c
$(CC) example /t4 /h /o example.t4h $(COPT)
```

This rule first defines the target as the compiled program `example.t4h`, which is dependent on the source file `example.c` and then specifies the command that must be invoked to build it.

The first rule in all makefiles is for the main target. Succeeding rules define sub-components of the main target, and are listed hierarchically.

Action strings

Action strings define the complete command line needed to recreate a specific file. The format is similar for all tools and consists of a call to the tool via a predefined macro, a fixed set of parameters, a list of command line options, probably also via a macro, and the output filename. (The output file is specified on the command line so that the rebuilt file is always written to the directory that contains the source.)

11.8.3 Delete rule

The delete rule directs make to remove all intermediate object files once the program has been built. It consists of a single labelled action string which invokes the host system 'delete file' command. Deletion is only performed if make is subsequently invoked with the `DELETE` option.

The delete rule is appended to the makefile by specifying the `imakef 'R'` option.

11.8.4 Editing the makefile

Makefiles created by the `imakef` tool can be edited for specific requirements. For example, new macros can be added and new rules defined for compiling and linking code written in other languages.

Adding options

`imakef` generates action strings which have the minimum of options for each tool. In most cases additional options are unnecessary or may be specified using compiler directives. To modify the set of default options for a particular tool simply edit the appropriate macro in the makefile.

For example, if the output of progress information is to be enabled for all invocations of the compiler, the compiler '`I`' option would be added to the macro which defines the standard combination of options for invoking the compiler. Alternatively a new macro containing only the '`I`' option could be defined and added to each compiler action string.

Re-running imakef

Once the set of options have been changed in the macros, it is useful to retain this set of options when `imakef` is run again. For this reason, `imakef` will check for the existence of a previous makefile. If one exists, it will re-use (in the new makefile) the set of macro definitions from the old one, plus any additional text up to a line marked `"IMAKEF CUT"`.

11.9 Error messages

imakef generates error messages of severities *Warning* and *Error*. Messages are displayed in standard toolset format.

Cannot have a makefile

The file specified on the command line is not one for which **imakef** can generate a makefile. **imakef** can only create makefiles for object files and bootable files.

Cannot open "*filename*" :*reason*

The file specified as the output file cannot be opened for writing by the program, for the reason given.

Cannot write linker command file

The linker command file cannot be opened for writing by the program.

Command line is invalid

An incorrect command line was supplied to the program. Check the syntax of the command and try again.

Error whilst reading

A file system error has occurred whilst reading the source.

#IMPORT references are illegal in configuration text

At the given line number in the file there is a reference to the **#IMPORT** directive, which is illegal for configuration source.

#INCLUDE may not reference a library

The **#INCLUDE** directive is being used to reference a file with the **.lib** extension.

#INCLUDE may not reference binary files

The **#INCLUDE** directive is being used to reference a file containing compiled code.

Incomplete compiler directive

At the given line number in the file there is an invalid compiler directive.

Library on PATH "*pathname*" also exists in the current directory

A library with the specified name has been found on the current search path and in the current directory.

Malloc failed

The program has failed while trying to dynamically allocate memory for its own use. Try using a transputer board with more memory. If the program is being run on the host it may be possible to increase the memory available using host commands.

Options are incorrectly delimited

The terminating bracket, which determines the options in a library build file, is missing at the given line number.

#SC references are illegal in configuration text

Applies to OCCAM modules only.

At the given line number in the file there is a #SC directive, which is illegal in configuration source code.

#SC, #USE may not reference source files

Applies to OCCAM modules only.

The directives #SC and #USE cannot be used to reference OCCAM source code.

Source file does not exist

The referenced source file does not exist on the system.

Target is not a derivable file

The specified file cannot be generated by the toolset.

Tree checking failed - no output performed

The tree of files has been found to be invalid and unusable for generating makefile. This message always follows a message indicating what is wrong with the tree. The most common reason for this error is the presence of cyclic references in the source.

"filename" unknown/illegal file reference

A compiler directive is attempting to reference the wrong type of file.

Writing file

A host system error occurred while the file was being written.

12 `imap` – memory mapper

This chapter describes the memory map tool `imap`. The tool takes the text output from the toolset compiler, linker and collector and gives the absolute addresses of the static variables for functions. The chapter begins with an introduction to `imap` and explains the command line syntax. `imap`'s output is described in some detail and an example is given. The chapter ends with a list of error messages.

12.1 Introduction

The `imap` tool takes as input memory map files output by the compiler, linker and collector. Command line options for these tools enable the user to specify that a memory map file is to be produced. `imap` collates the information from the different source files and puts it in a format suitable for output on the display screen. Alternatively the output from `imap` can be redirected to another output file as the user wishes. Memory maps may be generated for both single and multiprocessor transputer programs.

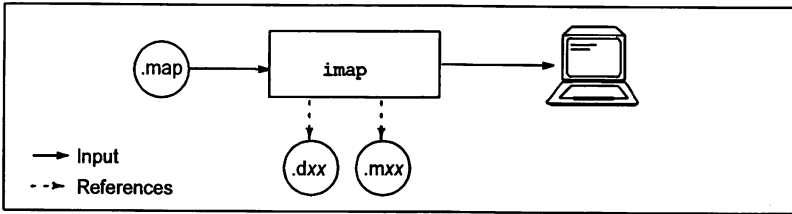
`imap` is invoked by supplying it with the name of a map file produced by the collector. The tool will automatically determine the names of map files produced by the linker and compiler, provided the naming convention for extensions has been adhered to, see section 12.2.1. Each tool generates a different level of information:

- Collector:** For each process on each processor, memory locations of code, static, heap, stack, invocation stack and vector space are listed.
- Linker:** For each process the offset in memory for code and static for individual modules which make up the process are listed.
- Compiler:** For each module listed in the linker output file the offset in memory for individual static items and functions are listed.

Where a particular category of information is not applicable to a language, this field will be left blank. Occam programs for instance do not use heap, so obviously such details are not generated for Occam.

Where the output files from the compiler and linker cannot be opened or parsed properly `imap` will insert a warning at the appropriate point in the output.

The operation of the map tool in terms of standard toolset file extensions is shown below. Output is sent to standard out, which is usually set to the display screen.



12.2 Running the map tool

To invoke the map tool use the following command line:

► **imap** *filename* { *options* }

where: *filename* is the name of the file containing the map output from the collector **icollect**. If there is no extension given, **.map** is assumed. Otherwise the file name is taken as given.

options is a list of the options given in Table 12.1

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Examples of use:

```
imap myprog
```

```
imap myprog.map
```

Both the above examples will cause **imap** to read the file **myprog.map**, generated by the collector.

Option	Description
A	Displays the list of symbols produced by the linker, including those symbols the linker identifies as not being used. This option will not override the 'R' option if it is used.
I	Displays progress information as <i>imap</i> processes information from the input files, such as the filenames of files as they are opened and closed.
O filename	Specifies an output file.
R	This option reduces the amount of detail generated by <i>imap</i> in two ways: <ul style="list-style-type: none"> the Module memory usage table only displays details for user modules i.e. 'USER' and 'SHARED_USER' processes. the Symbol table excludes those symbols containing a '%' character in their name. Such symbols are normally internal symbols e.g. C runtime library symbols.
ROM hex offset	This option is only applicable to, and must be specified for, code targetted at ROM. It enables a hexadecimal offset to be specified which represents the start address of the code in ROM. This offset will be added to the start address of any code which is to run in ROM, in <i>imap</i> 's output.

Table 12.1 *imap* command line options

12.2.1 Source files required by *imap*

Three different types of source file are read by *imap* and should therefore be made available. The files are in fact memory maps generated by the compiler, linker and collector. The appropriate command line option must be specified on each tool's command line including a filename for the map produced. The filename specified by the user must have the appropriate extension as indicated in table 12.2.

Extension	File description
.mxx	Map file output by the compiler. The characters 'xx' are determined by the 2nd and 3rd characters of the extension given to the compiler object file. For example if the compiler object file takes the default extension .tco, the information file is given the extension .mco.
.dxx	Map file output by the linker. The characters 'xx' are determined by the 2nd and 3rd characters of the extension given to the linker object file. For example if the linker object file takes the default extension .1ku, the information file is given the extension .dku.
.map	Map file output by the collector.

Table 12.2 Files extensions for *imap* source files

12.2.2 Re-directing `imap`'s output

`imap`'s output goes to standard out by default. To redirect it to an output file, use the '`o`' option and specify an output filename.

12.3 Output format

This section describes the format of the memory map produced by `imap`. An example output is given in section 12.4.

If the `imap` tool cannot find a linker or compiler output file, it will insert a warning message in place of the missing information. It will not produce a warning if the process or module comes from a library (such as the system process library).

12.3.1 imap memory map structure

- All processors used are listed. Details for each processor are as follows:
 - Processor name – if specified by the user in a configuration data file.
 - Processor Id. – a unique number which is assigned by the collector. These numbers start from zero, and increase by one for each processor.
 - Processor type e.g. T400, T805 etc.
- A list of processes, (in the same order as the collector output file). The following details are given for each process as appropriate:
 - Process Id. – a three digit number assigned by `imap`. Process numbers start at 000 and are incremented by 1 for each new process.
 - Process priority – `HIGH` or `LOW`
 - Process type – `USER`; `SHARED_USER`; `INITSYSTEM`; `SYSTEM` or `OVERLAYED_SYSTEM`. See table 12.3.
 - Process name, if specified by the user.
 - List of linked units or a library from which the process is made. For each linked unit or library the following details are included:
 - Name of the linked unit or library
 - The file offset in memory, expressed as a decimal number.
 - The offset from the start of the linked unit or library, at which code for that process begins, (expressed as a decimal number). In a linked unit this will usually be a very small number, pointing to just after the header, whereas in a library, where the code is selected from a number of related chunks of code, the offset may point to anywhere within the library.
 - A list of modules that make up the unit. The list is in the same order as the linker output file. Details include:
 - Module Id. – a three digit module number assigned `imap`.
 - TCOFF object filename.
 - Name of the source file from which the object was generated.
- A table of the memory blocks allocated to user processes. See 12.3.3.
- A table of the code and static memory blocks used by individual modules, including the section of memory that each block represents. See 12.3.4.
- A table of the memory blocks that non-user processes use. See 12.3.5.
- A table of symbols used by all of the processors. See 12.3.6.

12.3.2 Process types

Process type	Description
INITSYSTEM	A process used in the initialization of the transputer network.
OVERLAYED_SYSTEM	System process which is overwritten by other processes after it has been used.
SYSTEM	A process used in the initialization and running of the network.
USER	A user process.
SHARED_USER	A user process whose code can be used by more than one process.

Table 12.3 Process types

12.3.3 User processes

The table headed with "User processes" gives the start and end addresses and lengths of the various blocks of memory used by the user processes for that processor. The table is ordered by start address and is structured as follows:

- Process number or 'All' if it is the parameter data block, which is not associated to just one process.
- Memory block type – **stack**; **overhd**; **code**; **heap**; **static** or **param**. See table 12.4.
- Start address in hexadecimal.
- End address in hexadecimal.
- Length in decimal.

Block type	Description
stack	Used for workspace
overhd	Used for invocation stack
code	Used for code
heap	Used for heap
static	Used for static data
param	Used for the parameter data block

Table 12.4 Memory block types

12.3.4 Module memory usage

The table headed with "Module memory usage" gives the memory areas that are used by each module for code or static data. The table is ordered by start address and has the following format:

- Process i.d.
- Module i.d.
- Type (code or static)
- Name of the section that the area belongs to
- Start address in hexadecimal.
- End address in hexadecimal.
- Length of the area in decimal.

Examples of section names are `pri%text%base` and `text%base` for code, and `static%base` for static data.

If the 'R' command line option is used only details of user processes are shown.

12.3.5 Other processes

The table headed with "Other processes" is the same as the "User processes" table but for all the non-user processes. This table will not include an entry for the parameter data block.

12.3.6 Symbol table

The symbol table is alphabetically ordered by symbol name and gives the following information:

- Symbol name.
- Processor Id.
- Process Id.
- Module Id.
- Start address associated with symbol (in hexadecimal).
- Symbol type (see below).

The type field of the symbol table is either taken directly from the compiler map file, or is created by `imap`. In the latter case, the field will be enclosed in parentheses. This information is based on which section the symbol comes from. Refer to the compiler documentation for the meaning of items in this field that aren't enclosed in parentheses.

Note: command line options can be used to extend or limit the amount of symbol information generated. Normally `imap` only gives details of symbols used by the program; the 'A' option instructs `imap` to include unused symbols in the list. The 'R' option prevents details of internal symbols, such as those used by the runtime libraries, being listed.

12.4 Example

The following example, for a single processor program, was generated by the command:

```
imap test -r                                     (UNIX)
imap test /r                                    (MS_DOS and VMS)
```

Memory map for 'test1'

=====

Map for processor 0 (T800)

List of processes

P:000 - LOW priority INITSYSTEM process: 'Init.system'
From 'sysproc.lib' (offset 4969)

P:001 - LOW priority SYSTEM process: 'System.process.a.'
From 'sysproc.lib' (offset 13391)

P:002 - HIGH priority SYSTEM process: 'System.process.b'
From 'sysproc.lib' (offset 28774)

P:003 - LOW priority USER process:

From 'testt800.lku' (offset 2)

```
M:000 - Module 'testt800.tco' from 'testt800.c'
M:001 - Module '/inmos/prod/d4214b/libs/centry.lib' from 'tmp.occ'
M:002 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'exit.p8x'
M:003 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'virtual.tmp'
M:004 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'semprocs.tmp'
M:005 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'support.c'
M:006 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'chandeb.c'
M:007 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'plus.c'
M:008 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'newsem2.c'
M:009 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'printf.c'
M:010 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'stdioini.c'
M:011 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'tmp.s'
M:012 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'crun1.c'
M:013 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'crun2.c'
M:014 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ioprgnam.c'
M:015 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'iocmdlin.c'
M:016 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'iscmdlin.c'
M:017 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isbuf.c'
M:018 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ismisc.c'
M:019 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isversn.c'
M:020 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ctype.c'
M:021 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'clock.c'
M:022 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ptime.c'
M:023 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'atexit.c'
M:024 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ioinit.c'
M:025 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'agwalloc.c'
M:026 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'flushbuf.c'
M:027 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'misc.c'
M:028 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'fflush.c'
M:029 - Module '/inmos/prod/d4214b/libs/libc.lib' from '<fabricated>'
M:030 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'istatic.c'
```

Figure 12.1 imap example, screen 1 of 3

```

M:031 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'tmp.s'
M:032 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ioisatty.c'
M:033 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'tmp.s'
M:034 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'memcpy.c'
M:035 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'memmove.c'
M:036 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'memset.c'
M:037 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'fprintf.c'
M:038 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'math.c'
M:039 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'strcpy.c'
M:040 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'strlen.c'
M:041 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'fseek.c'
M:042 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'iofsz.c'
M:043 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isfseek.c'
M:044 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isftell.c'
M:045 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'clearerr.c'
M:046 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'ftell.c'
M:047 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'vfprintf.c'
M:048 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'writebuf.c'
M:049 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'iolseek.c'
M:050 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'iowrite.c'
M:051 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isfflush.c'
M:052 - Module '/inmos/prod/d4214b/libs/libc.lib' from 'isfwrite.c'

```

User processes:

Process	Type	Start	End	Length
P:003	stack	#80000154	#80000267	276
P:003	overhd	#80000268	#8000027B	20
P:003	code	#80000284	#80004E2F	19372
P:All	param	#80004E30	#80004FEF	448
P:003	static	#800052A0	#800054D9	570

Module memory usage:

Process	Module	Type	Section	Start	End	Length
P:003	M:000	code	text%base	#80000284	#800005C7	836
P:003	M:000	static	module%table%base	#800052A0	#800052A3	4
P:003	M:000	static	static%base	#80005300	#8000531B	28

Other processes:

Process	Type	Start	End	Length
P:002	stack	#80000154	#8000016B	24
P:002	overhd	#8000016C	#8000017F	20
P:002	code	#80000188	#800001E3	92
P:000	stack	#80004FF0	#8000505F	112
P:000	overhd	#80005060	#80005077	24
P:000	code	#80005080	#8000529F	544
P:001	stack	#800052A0	#800054B7	536
P:001	overhd	#800054B8	#800054CB	20
P:001	code	#800054D4	#80005E1F	2380
P:001	vector	#80005E20	#8000601F	512

Figure 12.2 imap example, screen 2 of 3

Table of symbols

Symbol name	Processor	Process	Module	Start	Type

C.ENTRY	0	P:003	M:001	#800005C9	(code)
_IMS_IsFileBuffer	0	P:003	M:005	#80005364	(static)
_IMS_StartTime	0	P:003	M:005	#8000532C	(static)
_IMS_SystemLink	0	P:003	M:005	#8000535C	(static)
_IMS_board_type	0	P:003	M:005	#800055A0	(static)
_IMS_closeptr	0	P:003	M:024	#80005A90	(static)
_IMS_ctype	0	P:003	M:020	#800058B8	(static)
_IMS_entry_term_mode	0	P:003	M:005	#80005330	(static)
_IMS_errno	0	P:003	M:005	#8000531C	(static)
_IMS_fclopseptr	0	P:003	M:010	#800055B0	(static)
_IMS_fdmapi	0	P:003	M:024	#80005A8C	(static)
_IMS_fflushptr	0	P:003	M:010	#800055B4	(static)
_IMS_heap_base	0	P:003	M:005	#80005348	(static)
_IMS_heap_front	0	P:003	M:005	#8000534C	(static)
_IMS_host_type	0	P:003	M:005	#8000559C	(static)
_IMS_huge_val	0	P:003	M:005	#80005334	(static)
_IMS_iob	0	P:003	M:010	#800055B8	(static)
_IMS_isbufsize	0	P:003	M:005	#800055A4	(static)
_IMS_last_recorded_wptr	0	P:003	M:005	#8000533C	(static)
_IMS_max_wptr_extent	0	P:003	M:005	#80005354	(static)
_IMS_os_type	0	P:003	M:005	#80005598	(static)
_IMS_printf	0	P:003	M:009	#80000DBA	(code)
_IMS_retnval	0	P:003	M:005	#80005328	(static)
_IMS_shared_area	0	P:003	M:005	#80005340	(static)
_IMS_stack_base	0	P:003	M:005	#80005350	(static)
_IMS_stack_limit	0	P:003	M:005	#80005344	(static)
_IMS_startenv	0	P:003	M:005	#80005320	(static)
calloc	0	P:003	M:025	#800020E1	(code)
clock	0	P:003	M:021	#80001B54	(code)
exit	0	P:003	M:023	#80001BFD	(code)
exit.p8x:B905FF7C	0	P:003	M:002	#80000284	(code)
fflush	0	P:003	M:028	#80002727	(code)
free	0	P:003	M:025	#800020C0	(code)
frexp	0	P:003	M:038	#80003600	(code)
ldexp	0	P:003	M:038	#80003580	(code)
longjmp	0	P:003	M:033	#80002925	(code)
main	0	P:003	M:000	#80000299	code
malloc	0	P:003	M:025	#80001FB5	(code)
memcpy	0	P:003	M:034	#80002940	(code)
memmove	0	P:003	M:035	#8000294C	(code)
memset	0	P:003	M:036	#80002A60	(code)
printf	0	P:003	M:037	#80003403	(code)
semprocs.tmp:3B081E0C	0	P:003	M:004	#80000284	(code)
strcpy	0	P:003	M:039	#800036D4	(code)
strlen	0	P:003	M:040	#80003744	(code)
test1	0	P:003	M:000	#80005300	data
test2	0	P:003	M:000	#80005304	data
test3	0	P:003	M:000	#80005308	data
testfunctionont800	0	P:003	M:000	#80000288	code
tolower	0	P:003	M:020	#80001A83	(code)
virtual.tmp:E5F06A7A	0	P:003	M:003	#80000284	(code)

Figure 12.3 imap example, screen 3 of 3

12.5 Error messages

This section lists each error message that can be generated by the memory map tool. Messages are in the standard toolset format which is explained in appendix A.

All open files are closed when an error is found and the tool halts without producing a map.

12.5.1 Serious errors

Filename input file cannot be parsed properly

The named file cannot be read by *imap*.

Cannot open collector's output file for reading

The collector map file specified on the command line cannot be found. Check that the extension used for the collector map file is in the correct format. See section 12.2.

Cannot open output file for writing

The output file cannot be opened for writing. May indicate a disk space problem or some other host system error.

Error parsing command line

The command line has the wrong syntax or a non-existent option has been specified.

Must specify input file

An input file must be specified.

Only single output filename allowed

More than one output filename has been specified.

Only single ROM offset value allowed

More than one ROM offset has been specified.

12.5.2 Fatal errors

Filename internal data structure failure or file corrupt

A source file used by *imap* has referenced something which cannot be found. This can occur when redundant map files are read by *imap* in error.

Filename out of heap space

There is not enough heap space to generate the memory map.

Unexpected end of file

A source file, read by `imap` has been corrupted. Regenerate compiler, linker and collector map files.

13 `iserver` — host file server

This chapter describes the host file server `iserver` which loads application programs onto transputer networks and provides runtime access to the host.

This document describes version 1.5 of the server. This is completely compatible with earlier versions but provides greater flexibility in the way transputers are accessed. In particular, the server can access transputer systems connected to a computer network (e.g. via Ethernet).

A summary of the new features of this version appears in Section 13.9.

13.1 Introduction

The host file server `iserver` provides three functions:

- Loading bootable programs onto transputer systems.
- A runtime environment for application programs, giving access to host services (e.g. file and terminal i/o).
- Controlled access to transputer systems; multiple users in a computer network can request a specific transputer system or a particular type of transputer. Access to each transputer system is granted to one user at a time for as long as required.

13.2 Loading programs

Before a program can be loaded into a transputer network it must be compiled and linked. Multi-transputer programs must also be configured for the transputer system they are to run on. The program is made bootable with the collector tool, `icollect`. The bootable file will generally have a `.bt1` file extension.

13.3 Host interface

Generally, transputer applications communicate with the host file server using the standard i/o libraries for the language being used. The library calls available and their parameters will be documented in the relevant language manual.

The communication with the host is based on a protocol, defined in appendix D. This protocol is used by the C, FORTRAN, and occam run time libraries to communicate with the host.

13.4 Access to transputer networks

Previous versions of the server required different code (and, therefore, a different executable) for every type of host interface. This version of the server provides support for all types of link interface in a single program. This means that when the server is run, it must be told what transputer systems are available and which type of interface to use for each.

User links

Each transputer system can be thought of as being connected to the host by a *user link*. User links are given descriptive names (known as *capabilities*) which are used by the server to find a suitable transputer system. Programs may be loaded onto transputer systems down user links, whilst operating system services are provided to the programs by communicating with the host file server via the user link. The names of the available user links, and the way in which the transputers are accessed, are defined in a *connection database* file.

More detail on user links, the connection database file, and accessing transputer networks is provided in section 13.7.

The session manager

The `iserver` guarantees unique access to a transputer system while it is running. The system is released when the server terminates. Often, the same resource needs to be used to run several programs (or run one program several times). For example, after running a program, it may be necessary to use the debugger — it would not be very useful if another user had started using the transputer between the program failing and the debugger starting.

The server's *session manager* allows access to a resource to be guaranteed for as long as required. The session manager is started with the 'SM' option and provides a simple command line interface. All the normal host operating system commands can be used as well as the `iserver` and session manager commands.

More detail on using the session manager are given in section 13.6.

13.5 Running the `iserver`

To run the host file server use the following command line:

► `iserver {options}`

where: *options* is a list of one or more options from Table 13.1.

If `iserver` is invoked with no options, help information is displayed, briefly explaining the command line arguments.

Some parameters can also be provided by the environment variables which are described in Section 13.5.2. These can be overridden by values provided on the command line.

Option	Description
SA	Analyses the root transputer and peeks 8K of its memory.
SB <i>filename</i>	Boots the program contained in the named file.
SC <i>filename</i>	Copies the named file to the root transputer link.
SE	Terminates the server if the transputer error flag is set or a control link error message is received.
SI	Displays progress information as the program is loaded.
SK <i>interval</i>	Specifies the number of seconds between attempts to access the resource.
SL <i>name</i>	Specifies the capability name.
SM	Invokes the session manager interface.
SP <i>n</i>	Sets the size of memory to peek on Analyse to <i>n</i> Kbytes.
SR	Resets the root transputer and its subsystem.
SS	Serves the link, i.e. provides host system support to programs communicating on the host link.
ST	All of the following command line is passed directly to the booted program as parameters.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. There must be at least one space between options. The case of letters in the parameters are not significant. Options may be in any order, except that no further options may appear after ST . Option ' SB <i>filename</i> ' is equivalent to ' SR SS SI SC <i>filename</i> '.	

Table 13.1 Host file server options

13.5.1 Examples of use

UNIX based toolsets:

```
icc hello
ilink hello.tco -f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS/VMS based toolsets:

```
icc hello
ilink hello.tco /f cstartup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

In this example **iserver** is executed to load and serve the bootable file **hello.btl** and to terminate on error. The example also shows the steps for compiling, linking, configuring and making the bootable file.

Note: this example assumes that the environment variable **TRANSPUTER** has been set to the name of the user link to be used.

13.5.2 Server environment variables

The server may obtain some parameters by inspecting environment variables. The following names are used:

TRANSPUTER Defines the capability (user link name) to be used by the server. May be overridden by the **SL** option.

ICONDB Defines the name of the connection database file used by the server.

ISESSION Defines the name of the session manager configuration file. The default is '`session.cfg`'.

13.5.3 Loading programs

Before a program can be loaded onto a transputer network it must be compiled, linked and made bootable.

Running a program using the *iserver* — option **SB**

The name of the file containing the bootable program is specified using the '**SB**' option. This resets the board and then copies the contents of the bootable file to the user link. If the file cannot be found an error is reported. When the program has been successfully loaded the server then provides host services to the program.

Note: Using the '**SB**' option is equivalent to using the **SR**, **SS**, **SI** and **SC** options together.

Sending data down a user link — option **SC**

To send data down a user link, or to load a program onto a board without resetting the root transputer, use the '**SC**' option — this simply copies the contents of the specified file to the user link. This should only be done if the transputer is running a program that can interpret the data sent down the link, or has already been reset. To reset the transputer subsystem use the '**SR**' option.

Running programs which do not use the server

To terminate the server immediately after loading the program use the '**SR**' and '**SC**' options together. This combination of options resets the transputer, loads the program onto the board, and terminates the server — the program on the transputer will continue running. This can be used to run a program which does not need to use the server facilities. If the program on the transputer attempts to access the server then it will deadlock until the server is run with the '**SS**' option.

Note: single transputer programs built with the collector's '**T**' option *cannot* be run in this way as the loader uses the server to read the value of the environment variable **IBOARDSIZE**. Configured programs (and programs built with the collector's '**T**' and '**M**' options) will perform no communication with the host other than that specified in the user's program.

Analyzing a transputer network — option **SA**

To load a board in analyze mode, for example when you wish to use the debugger to examine the program's execution, use the '**SA**' option to dump a section of the transputer's memory (starting from **MOSTNEG INT**). The default amount of memory to dump is 8 Kbytes, but this can be overridden with the '**SP**' option. The data is stored in an internal buffer which can be read by the `idump` tool when programs that use the root transputer are to be debugged.

Terminating the server

When the program sends a terminate command to the server (or some serious error occurs) the server will terminate if the session manager is not being used. If an error occurred an error message will be printed. If the session manager interface is being used, then control will return to the session manager.

13.5.4 Supplying parameters to a program

Any parameters on the command line following the '**ST**' option are passed as parameters to a booted program. This includes parameters that would normally be interpreted as `iserver` parameters. In addition, any text on the command line that is not recognized as a server parameter is also passed to a booted program.

13.5.5 Specifying the transputer resource — option **SL**

To specify the transputer resource to be used a capability name must be specified. The server may be given the capability on the command line using the '**SL**' option.

The capability may also be specified by the environment variable **TRANSPUTER**. This variable is overridden by a capability specified by the '**SL**' option.

The **SK** option can be used to specify how frequently the server should retry if the requested resource is not available.

13.5.6 Terminating on error — option **SE**

When debugging programs it is useful to force the server to terminate when the subsystem's error flag is set. To do this use the '**SE**' option. This option should only be used for programs written entirely in **OCCAM** and compiled in **HALT** system mode. If the program is not written entirely in **OCCAM** then the error flag may be set even though no error has occurred.

13.5.7 Terminating the server

To terminate the server press the host system break key. The server will either then terminate, returning to the host operating system prompt, or return to the session manager interface prompt if the server was invoked with the '**SM**' option.

13.5.8 Specifying the session manager configuration file

The session manager configuration file contains **iserver** commands for use by the session manager and may be customized for personal use.

The file is given by the environment variable **ISESSION**. If **ISESSION** is not set, then the filename '**session.cfg**' is used.

13.6 Using the session manager interface

The session manager provides a mechanism for guaranteeing unique access to a transputer system for as long as is required. Once the server terminates, or the system is released from within the session manager, access to the same system is no longer guaranteed.

13.6.1 Session manager commands

The session manager has a simple command line interface. There are a number of commands for managing the session such as **open** to select the transputer resource to be used, and **exit** to terminate the session. A complete list of session manager commands is given in table 13.2.

Command	Description
iserver parameters	Run as server to load network and provide host services. This command has most of the same options as the normal iserver command (except, of course, ' SL ' and ' SM ').
source filename	Read commands from a file
open capability	Release any held system and open a session with the named capability.
options parameters	Specify command line parameters to iserver command.
show	List user-defined commands.
help	List internal and user-defined commands.
exit or quit	Release any held system and exit the session manager.
user command	A user-defined command.
any other command	Any other command is passed directly to the host operating system.

Table 13.2 Session manager commands

13.6.2 The options command

The **options** command allows a set of standard options for the session manager's **iserver** command to be defined. Any parameters to the **options** command

are saved and added to any following **iserver** command. See the examples in sections 13.6.3 and 13.6.4.

With no parameters, the **options** command displays the parameters which are currently set. To remove parameters which have previously been set with the **options** command, use an empty string as the parameter, i.e.

```
options ""
```

13.6.3 The **iserver** command

The command **iserver** starts the server, from within the session manager, to load code onto the transputer system and provide host services to it. This command can be followed by any of the command line options described in section 13.5 (except 'SI' and 'SM' which are ignored). This allows programs to be run repeatedly on the same transputer system. When the program running on the transputer terminates (whether due to a normal termination, an error condition, or a user interrupt) control is returned to the session manager, without releasing the transputer system.

Note: the options to the session manager **iserver** command must be preceded by the appropriate option character as defined in table 13.1.

When an **iserver** command is executed, either directly or via a user-defined command, the parameters are built up from 3 parts in the following order:

- 1 The parameters supplied with the **options** command.
- 2 The parameters entered on the command line by the user.
- 3 The parameters from the command definition (if a user-defined command).

Example (UNIX based systems):

If the following commands are executed in the session manager:

```
# options -sb ika.btl
# iserver -si 42 tako.dat
```

Then the following **iserver** command line is generated:

<pre>iserver</pre>	<pre>-sb ika.btl</pre>	<pre>-si 42 tako.dat</pre>
	Parameters from the options command	Parameters from the command line

Example (MS-DOS/VMS based systems):

If the following commands are executed in the session manager:

```
# options /sb ika.btl
# iserver /si 42 tako.dat
```

Then the following **iserver** command line is generated:

iserver	<code>/sb ika.btl</code>	<code>/si 42 tako.dat</code>
	Parameters from the options command	Parameters from the command line

13.6.4 User-defined commands

In addition to the built-in commands of the session manager, new commands can be defined by the user. These commands define a set of parameters to be passed to the session manager **iserver** command, giving a shorthand for regularly used commands.

New commands are defined in the session manager configuration file, named in the environment variable **ISESSION**. The commands are defined as a set of parameters to the session manager's **iserver** command.

The format of the file is simple. Each command occupies a single line. The command name is the first word on the line. The rest of the line are the parameters to the **iserver** command that is to be substituted for the user-defined command.

As an example, suppose a program called **mytool** is normally booted onto a transputer with the **SE** and the **SB** options followed by user parameters for the tool. The normal **iserver** command line might look something like this:

```
iserver -se -sb /usr/tpbin/mytool.b8h parameters (UNIX)
iserver /se /sb /usr/tpbin/mytool.b8h parameters (MS-DOS/VMS)
```

To simplify this inside the session manager, the following lines would be put in the session manager configuration file:

```
mytool -se -sb /usr/tpbin/mytool.b8h (UNIX)
mytool /se /sb /usr/tpbin/mytool.b8h (MS-DOS/VMS)
```

Then, if the command '**mytool parameters**' is entered on the session manager command line it will be replaced with the command:

UNIX based systems:

iserver	<code>parameters</code>	<code>-se -sb /usr/tpbin/mytool.b8h</code>
	Parameters from the command line	Parameters from the command definition

MS-DOS/VMS based systems:

iserver	<code>parameters</code>	<code>/se /sb /usr/tpbin/mytool.b8h</code>
	Parameters from the command line	Parameters from the command definition

The server command line is built up as described above in section 13.6.3 so, if some extra parameters are defined with the **options** command they will be included as well. For example, the command sequence:

UNIX based systems:

```
options -si  
mytool parameters
```

MS-DOS/VMS based systems:

```
options /si  
mytool parameters
```

Would cause the following **iserver** command to be generated:

```
iserver -si parameters -se -sb /usr/tpbin/mytool.b8h (UNIX)  
iserver /si parameters /se /sb /usr/tpbin/mytool.b8h  
                                     (MS-DOS/VMS)
```

Running the debugger from the session manager

One important use of user defined commands in the session manager is to allow the debugger to be run. This is done by defining a command such as the following in the session manager configuration file.

```
idebug -se -sb /usr/local/D5214/itools/idebug.bt1 (UNIX)  
idebug /se /sb /usr/local/D5214/itools/idebug.bt1  
                                     (MS-DOS/VMS)
```

The exact form of this command will depend on which toolset is being used and the type of program being debugged. For more details on **idebug** command line options, see chapter 4.

13.6.5 Host OS commands

If a command is not one of the session manager's internal commands and not a user-defined command then it is passed to the host's operating system for execution.

13.7 Connecting transputers to computer networks

Transputer systems may be connected to the host computer in a number of ways. For example, they may be connected directly to the host via a board such as the IMS B008 motherboard, or via Ethernet using an IMS B300 TCPlink box.¹

Each connection, however implemented, is treated in the same way. Programs can be loaded onto a transputer system via any of these connections and that program can then gain access to host facilities and communicate with the user via the same connection. Each of these connections is known as a user link.

13.7.1 Capabilities

User links are identified to users (and tools) by name. Each user link has one or more names, known as *capabilities* — the names should be chosen to indicate to

1. It is also possible to remotely access a transputer directly connected to another (remote) host. This requires the host connection server (HCS) to be running on the remote host — currently the HCS software is only available for PCs.

a user what type of transputer system is connected to that user link. Examples of capability names that might be used are **B008**, **T801+2MB**, or **Grid:10x10**.

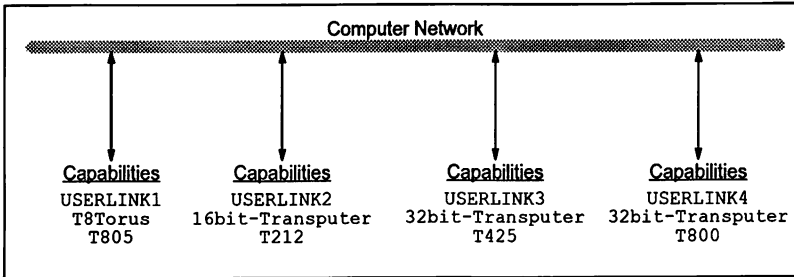


Figure 13.1 Capabilities for user links

If access to a transputer system is desired, one of the capability names of a user link is given to the **iserver**. If a user link of that name is free then unique access to that user link is granted until the server terminates. If the session manager interface of the server is used (see Section 13.6), then unique access to the user link can be maintained for as long as required. If the capability specified is not unique then the **iserver** searches for the first free user link on the local machine with that capability. If that capability is not available, remote machines are then searched. For this reason each user link will normally have several capabilities, at least one of which will be unique in the network.

Figure 13.1 shows an example computer network with four user links. Each user link has several capabilities. If a specific user link is to be used then the capability **USERLINK1**, **USERLINK2** etc. should be used. If that user link is free unique access to it will be given. If all that is required is a 32 bit transputer (to run a tool, for instance) then the capability **32bit-Transputer** could be used; connection to either of user links 3 or 4 could then be made, assuming that one of them is free.

When the **iserver** is run it is given the name of a capability with the '**SL**' option or the environment variable **TRANSPUTER**. The server finds the information it needs to access that transputer network from the connection database file.

13.7.2 The connection database

Each transputer system available is described in the connection database file. The **iserver**, when given the name of a transputer system, uses the connection database to determine how to access that system. The name of the connection database file is defined by the environment variable **ICONDB**.

The connection database is a text file which contains a description of all the available capabilities. In a PC development system, for example, the connection database may contain only a single entry — the transputer board which is installed in the PC. In a multi-user development environment, such as a networked Sun workstation, the database may contain a number of capabilities representing

transputer links connected directly to the host or accessible via Ethernet. In this case the connection database will be set up by the system administrator.

There may be several entries in the connection database with the same capability name. In this case, when a user runs **iserver** specifying this capability, the first system with that capability name which is not already in use will be allocated to the user.

The connection database must exist for **iserver** to be able to access a transputer. The server uses the environment variable **ICONDB** to locate the connection database file.

Examples and further details of the contents of the connection database are given in section 13.8.

13.7.3 Using a specific node

It is possible to request that a capability on a specific network node is used (note that 'node' in this context may be an IMS B300 Ethernet interface). To do this the character '@' is added after the capability name, followed by the name of the node. When this is done the named node is contacted directly. If the specified capability is not free on that node then the request will fail.

For example, to use the capability **T4-torus** on the node **pw11heli** the name **T4-torus@pw11heli** would be specified to the server.

The special name **localhost** is defined as being the the local host (alternatively, the local host's name may be used directly). If the local host is specified then only local user links may be used. If no suitable user link is found locally the request will fail.

13.8 The connection database

This section gives more detail about the connection database file that provides information to the server about the available transputer networks.

13.8.1 Connection databases

A connection database file must be available on your host before you can gain access to any user links. The connection database describes the transputer systems available on your network. The format of the connection database file is described below, in section 13.8.2.

In a single-user system such as a PC-compatible development system, there may only be a single entry in the connection database file – this will describe the transputer board installed in the PC. A typical entry for an IMS B008 board in a PC might look like this:

```
|B008|T|localhost|#150|b004||B008 with 3 x T805|
```

In a larger development environment, there may be many entries describing all the transputer systems in the entire network of development systems. In this case, the connection database file will usually be created and managed by the system administrator.

Capability names

There may be a number of different user links with the same name so that similar transputer systems can have the same capability. For example, all the user links which are connected to transputers which can run transputer based tools could be given the capability `ToolHost`.

Similarly, there may be more than one entry for each transputer system in order to give multiple names to the same resource. This allows a transputer network with a 30MHz T805 transputer, for example, to be described with the capabilities `ToolHost` (for user who want to use any transputer that can run transputer based tools) and `T805-30` (for those who need to use that specific type of transputer).

13.8.2 Connection database format

Connection database files are ASCII text files. It can contain four types of line:

- Blank lines — these are ignored.
- Lines starting with a '#' character are comment lines, and are ignored.
- Lines starting with a '|' character are record lines, containing connection data.
- Lines starting with a '*' character are continuation lines, used when a record line will not conveniently fit on one line.

Records in the database are made up of eight fields, in a fixed order. Fields are separated by a '|' character. Records may be broken over a number of lines if required, but may only be broken between fields. In this case a '*' character is used as the field separator, and must appear at the end of the line to be continued, and as the first character on the continuation line. The characters '|' and '*' are not permitted within any field. Trailing spaces in a field are ignored.

The fields are shown in Table 13.3. There are two field types, *String* and *Boolean*. A *String* is one or more ASCII text characters and a *Boolean* is a single ASCII character, either 'f' or 'F' for *false* or 't' or 'T' for *true*.

Field	Type	Description	Example
Capability	<i>String</i>	Capability (Name) of user link	T8-grid
IsWorking	<i>Boolean</i>	True if the user link is available	T
Machine	<i>String</i>	Network name of host machine for that user link (<code>localhost</code> if local)	pw1lheli
Linkname	<i>String</i>	Name of link connected to user link	/dev/bxvi0
Linkdev	<i>String</i>	Type of device providing user link	B016
Mmsfile	<i>String</i>	Reserved for future use	
Mmslink	<i>String</i>	Reserved for future use	
Description	<i>String</i>	Comment describing user link	T805 Square Grid

Table 13.3 Connection database record fields

13.8.3 Example connection databases

PC development system

The first example shows how 2 PC add-in boards in the same system could be described. Each board has two names: a common name (B008) to allow a user to access whichever user link is available, and a unique name to enable the user to specify which transputer system they wish to use (T400 or T805).

```
# The following resource describes a B008 with a T400.
# The link device is at address #150.
```

```
|T400 |T|localhost|#150|b004|||B008 board with T400 + 2MB|
|B008 |T|localhost|#150|b004|||B008 board with T400 + 2MB|
```

```
# The following resource describes a B008 with a T801.
# The link device is at address #200.
```

```
|T805 |T|localhost|#200|b004|||B008 board with T801 + 2MB|
|B008 |T|localhost|#200|b004|||B008 board with T801 + 2MB|
```

Sun workstation

This example shows how the various boards that can be connected to a Sun workstation directly, can be described in a connection database file.

```
# Sample connection database.
# The device names and addresses supplied are the default names
# and/or addresses suggested in the installation section of the
# board's manual.

# The following resource allows access to a B011 connected to this
# host. The resource name is "B011" and the device is accessed
# at address 0x800000.

|B011|T|localhost|0x800000|b011|||Description of board|

# The following resource allows access to a B014 connected to the
# host. The resource name is "B014" and the B014 device driver is
# accessed via "/dev/bxvi0".

|B014 |T|localhost|/dev/bxvi0|b014|||Description of board|

#
# The following resource allows access to a B016 connected to the
# host. The resource names are "B016" and the B016 device drivers
# are accessed via "/dev/bxvi0" through to "/dev/bxvi3".

|B016 |T|localhost|/dev/bxvi0|b016|||Description of board|
|B016 |T|localhost|/dev/bxvi1|b016|||Description of board|
|B016 |T|localhost|/dev/bxvi2|b016|||Description of board|
|B016 |T|localhost|/dev/bxvi3|b016|||Description of board|
```

IMS B300

The following example shows how four transputer systems available via Ethernet could be described.

```
# Connection database for an IMS B300 with the node name 'billy'

# A T425 connected to link 2 of the B300
|HOSTLINK0 |T|billy|2|tcp|||T425+1M|
|TA         |T|billy|2|tcp|||T425+1M|
|T425       |T|billy|2|tcp|||T425+1M|

# A T805 connected to link 3 of the B300
|HOSTLINK1 |T|billy|3|tcp|||T805+2M|
|TA         |T|billy|3|tcp|||T805+2M|
|T805       |T|billy|3|tcp|||T805+2M|

# Another T805 connected to link 0 of the B300
|HOSTLINK2 |T|billy|0|tcp|||T805+2M|
|TA         |T|billy|0|tcp|||T805+2M|
|T805       |T|billy|0|tcp|||T805+2M|

# A small network connected to link 1 of the B300
|HOSTLINK3 |T|billy|1|tcp|||T800+4M + T805+2M + T222+60K|
|TA         |T|billy|1|tcp|||T800+4M + T805+2M + T222+60K|
|T800       |T|billy|1|tcp|||T800+4M + T805+2M + T222+60K|
```

13.9 New server features

This section summarizes the main differences between version 1.5 of the *iserver* and previous versions.

The new features are:

- The addition of a session manager user interface.
- A connection manager has been added, and capability names are used instead of link names. There is no longer a default name.
- Some new command line options have been added.
- User interrupt behavior has changed.
- Exit codes have changed.
- New error codes have been added.
- Stream identifiers are validated.
- Support for record structured files has been added.

These changes are described in more detail in the following sections.

13.9.1 Session manager

This is a simple user interface that provides control of access to shared transputer resources. It will provide unique access to a specified transputer resource (if available) for as long as required.

13.9.2 Connection manager

The connection manager provides transparent access to both remote and local transputer resources. Resources are identified by a “capability name” and, optionally, the name of the host to which the resource is connected.

13.9.3 New command line options

Three new command line options have been added:

SK Retry the connection at intervals.

SM Invoke the session manager interface.

ST All arguments which follow are *not iserver* arguments and will be passed to the application. Note that this is a significant change to the way that the *iserver* parses its command line. Existing command files or shell scripts may need to be changed.

13.9.4 User interrupt

Behavior on user interruption depends on how the server is being run. If the session manager interface is being used, then the server returns to the session manager interface. If the session manager is not being used then the server terminates.

13.9.5 Exit codes

This version of the `iserver` makes it possible to distinguish between the various causes of termination of the server, such as user break, error flag set etc. Appendix D provides full details of the exit codes.

13.9.6 Error codes

Server operations now return a range of error codes to indicate the cause of a failure. Checks are now made to ensure that operations are supported, a particular transputer system is available etc.

Appendix D provides details of `iserver` error codes.

13.9.7 Stream identifier validation

Checks have been added to the server to validate all stream identifiers. Earlier versions of the server assumed that a stream identifier would always be valid.

13.9.8 Record structured file support

Support for record structured files has been added for all supported hosts. Supported formats are formatted sequential, unformatted sequential, formatted direct and unformatted direct. See Appendix D for full details.

13.10 Error messages

A list of possible error messages which `iserver` may produce follows. In some cases, these messages may be followed by an extra message giving additional information; these are listed below in section 13.10.1.

Aborted by user

The user interrupted the server, by pressing `Ctrl-C` or `Ctrl-Break` .

Boot filename is too long, maximum size is *number* characters

The specified filename was too long. *number* is the maximum size for filenames.

Cannot find boot file *filename*

The server cannot open the specified file.

Command line too long (at *string*)

The maximum permissible command line length has been exceeded. The overflow occurred at *string*.

Copy filename is too long, maximum size is *number* characters

The specified filename was too long. *number* is the maximum size for filenames.

Error flag raised by transputer

The program has set the error flag. Debug the program.

Expected a filename after –SB option

The 'SB' option requires the name of a file to load.

Expected a filename after –SC option

The 'SC' option requires the name of a file to load.

Expected a name after –SL option

The 'SL' option requires a link name or address.

Expected a number after –SP option

The 'SP' option must specify the number of Kbytes to peek.

Failed to allocate CoreDump buffer

The 'SP' option was used but the server was unable to allocate enough memory to allow the transputer's memory to be copied.

Failed to analyse root transputer

The link driver could not analyze the transputer.

Failed to reset root transputer

The link driver could not reset the transputer.

Reset and analyse are incompatible

Reset and analyze options cannot be used together.

Timed out peeking word *number*

The server was unable to peek the transputer.

Transputer error flag has been set

The program has set the error flag. Debug the program.

Unable to access a transputer

The server was unable to gain access to a link. This occurs when the link address or device name, specified either with the SL option or the **TRANSPUTER** environment variable, is incorrect or does not exist. This message will be followed by one of the messages listed below.

Unable to free transputer link

The server was unable to free the link resource because of a host error. The reason for the error will be host dependent.

Unable to get request from link

The server failed to get a packet from the transputer because of some general failure.

Unable to write byte *number* to the boot link

The transputer did not accept the file for loading. This can occur if the transputer was not reset or because the file was corrupted or in incorrect format.

13.10.1 Additional error messages

The following messages provide additional information to accompany error messages from the server.

: no environment variable ICONDB

There is no environment variable ICONDB.

: can't open connection database file [...]

The file specified in the environment variable ICONDB cannot be accessed.

connection database, file [...], at line [...]

→ premature end of file

The database file is corrupt, a record line is not complete.

connection database, file [...], at line [...]

→ premature end of file, looking for field {...}

The database file is corrupt. A record line is not complete; the field {...} does not exist.

connection database, file [...], at line [...]

→ expecting continuation character

Line [...] of the database file is corrupt. The record was not complete, a field is missing and there was no continuation indicating the record is continued on the next line.

connection database, file [...], at line [...]

→ expecting continuation character at start of line, looking for field {...}

Line [...] of the database file is corrupt. The previous line ended with a continuation character – a continuation was expected to start the current line.

connection database, file[...], at line[...]

→ can't start a line with continuation, looking for field {...}

Line [...] of the database file is corrupt. A record line started with a continuation character (it should start with a field separator). The {...} field was expected.

connection database, file[...], at line[...]

→bad field separator, looking for field {...}

Line [...] of the database file is corrupt; the field was illegal.

connection database, file[...], at line[...]

→ field {...} cannot be null

Line [...] of the database file is corrupt. The field {...} contained a null value (this is illegal).

connection database, file[...], at line[...]

→ illegal boolean value, looking for field {...}

Line [...] of the database file is corrupt. The field {...} should contain a boolean value.

connection database, file[...], at line[...]

→ illegal linkdev field – unknown method

Line [...] of the database file is corrupt. The Linkdev field should contain a link method value.

14 *isim* — T425 simulator

This chapter describes the T425 simulator tool *isim* that allows programs to be run and tested without transputer hardware. The chapter explains how to invoke the tool and describes the simulator commands that allow the simulated program to be debugged interactively.

14.1 Introduction

The simulator can run any transputer program that would run on a single IMS T425 mounted on a normal transputer evaluation board and supported by a host running *iserver*. No transputer hardware is required unless you have an MS-DOS host, in which case *isim* does require a 32-bit transputer processor. This is due to the memory requirements of *isim*.

Because the simulator runs the same code that would be loaded onto a real transputer, any program that runs satisfactorily in the simulator will run on an IMS T425. Because all 32-bit transputers are compatible at the source level, the same program can also be run on any IMS 32-bit processor after recompiling for the correct processor type.

The simulator also provides a reduced set of debugging facilities similar to those of the debugger Monitor page. Additional features provided by the simulator are the ability to set break points at simulated transputer addresses and to single step the program. The program should be loaded into memory (using the `[G]`, `[J]` or `[P]` commands) before breakpoint debugging facilities are used. This ensures that breakpoints are not overwritten during the booting phase.

The simulator can also be used to familiarize new users with transputers and transputer programming, and as a teaching aid.

14.2 Running the simulator

To run the simulator use the following command line:

► `isim program [programparameters] {options}`

where: *program* is the program bootable file.

programparameters is a list of parameters to the program. The list of parameters may follow the *isim* 'N' option and parameters must be separated by spaces. See section 14.2.1.

options is a list of options from Table 14.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be given in any order.

Options may be entered in upper or lower case and can be given in any order.

Only one filename may be given on the command line.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
B	Batch mode operation. The simulator runs in line mode i.e. full display data is not provided. Commands are read in from the input stream e.g. the keyboard and executed. The commands are not echoed to the output stream e.g. the display screen, as they are executed.
BQ	Batch Quiet mode. The simulator automatically executes the program specified on the command line and then terminates. If an error occurs, the appropriate message will be displayed. The debugging facilities of the simulator are not available in this mode.
BV	Batch Verify mode. Similar to batch mode, except that the commands and prompts displayed when running the simulator in interactive mode are echoed to the output stream e.g. the display.
I	Displays information about the simulator as it runs.
N	No more options for the simulator. Any options entered after this option will be assumed to be program parameters to be passed to the program running on the simulator.

Table 14.1 *isim* options

14.2.1 Passing in parameters to the program

Program parameters can be passed to programs which are simulated on any host. Parameter passing is equivalent to running a transputer bootable program using *iserver*.

isim will normally parse the command line and any options it recognizes as its own will not be passed to the user program. In cases where options are required for a user program which clash with one of the *isim* options the 'N' option can be used. After the 'N' option *isim* ceases parsing the command line for its own options; the remainder of the command line is simply passed through to the user program.

14.2.2 Example of use

```
isim hello.btl
```

This invokes the simulator on the 'Hello World' program.

When first invoked the simulator enters the debugging environment. To start the program invoke the 'G' command. The program then runs until it completes successfully, a runtime error occurs, or a break point is reached. If an error occurs the processor halts, the error flag is set, and the program can be debugged using commands to examine memory and registers.

When invoked with the 'BQ' option (Batch Quiet) the simulator immediately runs the program and does not enter the debugging environment.

14.2.3 ITERM file

The simulator reads the ITERM file to determine how to control the terminal screen and to map a few simulator commands. The ITERM file must be defined in the host environment variable **ITERM**.

14.3 Monitor page display

The simulator Monitor page is similar to that of the debugger, which is described in chapter 4. Data displayed at the simulator Monitor page includes:

Ip tr	Contents of instruction pointer (address of the <i>next</i> instruction to be executed).
Wp tr	Contents of workspace pointer.
Error	Status of error flag.
Halt On Error	Status of halt on error flag.
Fp tr1	Pointer to the front of the low priority active process queue. If 'jump 0' breaks are enabled the letter B is displayed after the pointer value.
Bp tr1	Pointer to the back of the low priority active process queue.
Fp tr0	Pointer to the front of the high priority active process queue.
Bp tr0	Pointer to the back of the high priority active process queue.
TP tr1	Pointer to the low priority timer queue. If the timer is disabled the letter x is displayed after the pointer value.
TP tr0	Pointer to the high priority timer queue.
Note: If Wp tr contains the most negative address value, it will be described as 'invalid'. This normally means that no process is executing in the simulator (for example, the program may have become deadlocked).	

The Monitor page also displays the last instruction executed, a summary of Monitor page commands, and, if an error has occurred, the cause of the error.

14.4 Simulator commands

All simulator commands are given at the Monitor page. Many of the commands are similar to those of the `idebug` Monitor page, however, there are a number of implementation differences. Full descriptions of the commands are given in the following sections.

14.4.1 Specifying numerical parameters

Some simulator commands require numerical parameters, such as addresses. These can be specified as simple expressions in decimal or hexadecimal format. Expressions can be the sum of two expressions, the result of subtracting one expression from another, or constants. Constants that can be specified are: `Areg`, `Breg`, `Creg`, `Iptr`, `Wptr`, decimal constants, hexadecimal constants, or abbreviated hexadecimal constants.

Hexadecimal constants are specified using the prefix `#`. Abbreviated hex constants can be created by prefixing the sequence of hex digits with `'%`', which assumes the hexadecimal prefix `'8000 . . . '`. For example, the abbreviation `'%F8A'` is interpreted as the hex number `'8000F8A'`.

14.4.2 Keys mapped by ITERM

Several commands for controlling the display are mapped to specific keys by the ITERM file. Key mappings for specific terminal keyboards can be found in the Delivery Manual.

HELP	Displays help information.
REFRESH	Re-draws the screen.
FINISH	Quits the simulator.
PAGE UP	Scrolls the current display.
PAGE DOWN	Scrolls the current display.
▲, ▼	Scrolls the current display.

14.4.3 Command summary

Key	Meaning	Description
A	ASCII	Displays a portion of memory in ASCII.
B	Break points	Breakpoint menu.
D	Disassemble	Displays transputer instructions at a specified area of memory.
G	Go	Runs (or resumes) the program.
H	Hex	Displays a portion of memory in hexadecimal.
I	Inspect	Displays a portion of memory in any occam type.
J	Jump into program	Runs (or resumes) the program. Same as G.
L	Links	Displays <code>Iptr</code> and <code>Wptr</code> for processes waiting for input or output on a link, or for a signal on the Event pin.
M	Memory map	This option is not supported for the current toolset.
N	Create dump file	Creates a core dump file.
P	Program boot	Simulates a program 'boot' onto the transputer.
Q	Quit	Quits the simulator.
R	Run queue	Displays <code>Iptr</code> and <code>Wptr</code> for processes on the high or low priority active process queues.
S	Single step	Executes the next transputer instruction.
T	Timer queue	Displays <code>Iptr</code> , <code>Wptr</code> , and wake-up times for processes on the high or low priority timer queues.
U	Assign register	Assigns a value to a register.
?	Help	Displays help information.

Table 14.2 Simulator commands

14.4.4 Command descriptions





A — ASCII

This command displays a segment of transputer memory in ASCII format, starting at a specific address. If no address is given the default address `Wptr` is used. Specify a start address after the prompt:





(Start address (`Wptr`)) ?

Either press `RETURN` to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

The memory is displayed in blocks of 13 rows of 32 ASCII bytes, each row preceded by an absolute address in hexadecimal. Bytes are ordered from left to right in each row. Unprintable characters are substituted by a full stop.

The , , , and  keys can be used to scroll the display.

– Breakpoints

Sets, displays, and cancels break points at specified memory locations or procedure calls. The program should be loaded into memory (using the ,  or  commands) before this command is used to set breakpoints. (The  command may also be used prior to this command, to determine where to set breakpoints).

The command displays the Breakpoint Options Page:

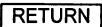
Breakpoint Options Page

- 1) Set breakpoint at Address
- 2) Display breakpoints
- 3) Cancel breakpoint at Address

Select Option?

Options are selected by entering one of the single digit commands. The following prompts are displayed depending on the command selection:

Command	Prompt
1	(break address) ?
3	(break address (ALL))

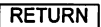
Pressing  with no typed input in response to command 1 cancels the option; in response to command 3, it causes *all* breakpoints to be cancelled.

After each breakpoint command the user is returned to the simulator command prompt.

– Disassemble





The Disassemble command disassembles memory into transputer instructions. Specify an address at which to start disassembly after the prompt:

(Start address (Ip_{tr})) ?

Either press  to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

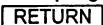
The memory is displayed in batches of thirteen transputer instructions, starting with the instruction at the specified address. If the specified address is within an instruction, the disassembly begins at the start of that instruction. Where the preceding code is data ending with a transputer 'pfix' or 'nfix' instruction, disassembly begins at the start of the pfix or nfix code.

Each instruction is displayed on a single line preceded by the address corresponding to the first byte of the instruction. The disassembly is a direct translation of memory contents into instructions; it neither inserts labels, nor provides symbolic operands.

The , , , and  keys can be used to scroll the display.

– Go

Starts the program, or continues running the program after a breakpoint or error has been encountered. The program will run until it completes successfully, sets the error flag, or reaches a break point.

To start the program, specify a break point address after the following prompt and press :


(break point address)

The default is not to set a break point.

– Hex

The Hex command displays memory in hexadecimal. Specify the start address after the prompt:

(Start address (Wptr) ?

Press  to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h. . . h'. If the specified start address is within a word, the start address is aligned to the start of that word.

The memory is displayed as rows of words in hexadecimal format. Each row contains four words of eight hexadecimal digits, with the most significant byte first.

Words are ordered left to right in the row starting from the lowest address. The word specified by the start address is the top leftmost word of the display.

The address at the start of each line is an absolute address displayed in hexadecimal format.

I – Inspect

Displays a portion of memory in any occam type – as defined in the 'occam 2 Reference Manual'.

The Inspect command can be used to inspect the contents of an entire array.

Specify a start address after the prompt:

(Start address (Wptr)) ?

Either press **RETURN** to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

When a start address has been given, the following prompt is displayed:

```

    Typed memory dump
0 - ASCII
1 - INT
2 - BYTE
3 - BOOL
4 - INT16
5 - INT32
6 - INT64      (Not implemented)
7 - REAL32     (Not implemented)
8 - REAL64     (Not implemented)
9 - CHAN

```

Which occam type ?

Give the number corresponding to the type you wish to display or press **RETURN** to accept the default type. Initially the default will be HEX; for subsequent use of the command the default takes the value of the last selected type.

ASCII arrays are displayed in the format used by the Monitor page command 'ASCII'. Other types are displayed both in their normal representation and hexadecimal format.

The memory is displayed as thirteen rows of data. The address at the start of each line is an absolute address displayed as a hexadecimal number. The element specified by the start address is on the top row of the display.

Start addresses are aligned to the nearest valid boundary for the type, that is: **BYTE** and **BOOL** to the nearest byte; **INT16** to the nearest even byte; **INT**, **INT32** and **CHAN** to the nearest word.

J – Jump into program

Same as **G** – starts or continues running the program.

L – Links

Displays information about simulated links.

The Links command displays the instruction pointer **Ip**, workspace descriptor **Wdesc** and priority, of the processes waiting for communication on a link, or for a signal on the Event pin. If no process is waiting, the link is described as 'Empty'. Link connections on the processor, and the link from which the processor was booted are also displayed.

The format of the display is similar to the following:

```
Link 0 out Ip: #80000256 Wdesc: #80000091 (Lo)
Link 1 out Empty
Link 2 out Empty
Link 3 out Empty
Link 0 in Empty
Link 1 in Empty
Link 2 in Empty
Link 3 in Empty

Link 0 connected to Host
Links 1, 2, 3 not connected

Booted from link 0
```

M – Memory map

This option is not applicable to the current version of **isim**, if used the following message will be displayed:

```
Memory Map Invalid
```

N – Create dump file

Creates a core dump file from which the program can be debugged off-line. The name of the file and the number of bytes to write must be specified. A file extension is not required and should not be specified. The dump file is automatically given the **.dump** extension.

P – Program boot

Loads the program into transputer memory ('boots the program') so that debugging can start at beginning of the application program without stepping through bootstrap loading code. The program is loaded into memory but is not automatically run. This command can only be used prior to executing any other instructions.

Q – Quit

Quits the simulator, and returns to the host operating system.

R – Run queues

This command displays `Iptrs` and `Wdescs` for processes waiting on the processor's active process queues. If both high and low priority front process queues are empty, the following message is displayed:

Both process queues are empty

If neither queue is empty, you are required to specify the queue:

High or low priority process queue ? (H,L)

Type 'H' or 'L' as required. If only one queue is empty `isim` displays the non-empty queue.

The , , `PAGE UP`, and `PAGE DOWN` keys can be used to scroll the display.

S – Single step transputer instruction

This command executes the transputer instruction pointed to by `Iptr`. By repeating the command the user may single step through the program, observing the changes to the process queues and registers, as the display is updated.

T – Timer queue

This command displays `Iptrs`, `Wdescs`, and wake-up times for processes waiting on the processor's timer queues. Prompts and displays are similar to those for the Run queue command.

U – Assign

Assigns a value to a register, `Iptr` or `Wptr`. To assign a value, specify the register by name (abbreviations are permitted), and give a value to be assigned to the register. This enables the program to be re-run (using `G` or `J`) with alternative values in the registers.

`HELP`

? — Help

Lists the available simulator commands.

`REFRESH` – Refresh

Refreshes the screen.

FINISH – Quit

Quits the simulator, and returns to the host operating system.

The **▲**, **▼**, **PAGE UP**, and **PAGE DOWN** keys can be used to scroll the display.

14.5 Batch mode operation

isim can be run in batch mode by setting up the environment variable **ISIMBATCH**. If this variable is defined on the system **isim** automatically selects batch mode operation.

14.5.1 Setting up ISIMBATCH

ISIMBATCH is set up on the system as an environment variable using the appropriate command for your host system.

VERIFY and **NOVERIFY** modes which enable and disable the output of input commands and user responses are defined by setting a value for **ISIMBATCH**. In MS-DOS the command to use is the **set** command. For example:

```
C:\ set ISIMBATCH=VERIFY
C:\ set ISIMBATCH=NOVERIFY
```

In UNIX the equivalent command is **setenv** and on VMS systems the command to use is **define**. Details of how to use these commands can be found in the user documentation for your system.

14.5.2 Input command files

In batch mode **isim** is driven from a command script containing simulator commands and responses to prompts. All prompts by **isim** must be followed by a valid response.

14.5.3 Output

Output can be written to a log file or displayed at the terminal. Input and output streams can be assigned to files or the user's terminal by commands on the host.

isim can be set up to operate in **VERIFY** or **NOVERIFY** mode by setting a different values for **ISIMBATCH**. In **VERIFY** mode all prompts and user responses are included in the output.

14.5.4 Batch mode commands

Batch mode simulator commands 'A' through 'U' are the same as the interactive debugging commands. Two additional commands generate special batch mode output:

Key	Meaning	Description
?	Query state	Displays values of registers and queue pointers.
.	Where	Displays next <i>Ip</i> tr and transputer instruction.

[?] – Query state

Displays information about the processor state, including current values of registers, queue pointers, and error flag status. For example:

```

Processor state
Iptr          #80000070
Wptr          #800000C8
Areg          #80000070
Breg          #800000C8
Creg          #80000010
Error         Clear
Halt on Error Set
Fptr1 (Low    #00000000
Bptr1 queue) #00000000
Fptr0 (High   #00000000
Bptr0 queue) #00000000
Tptr1 (timer  #2D2D2D2D
Tptr0 (queues #2D2D2D2D

```

[.] – Where

Displays the *Ip*tr of the next instruction to execute and a disassembly of that instruction. For example:

```
Ip
```

tr #80000070. Low Priority, Next Instruction : ajw 42 - #2A

14.6 Error messages

Cannot open bootfile '*filename*'

The file containing the code to be run could not be opened or could not be found.

Environment variable 'IBOARDSIZE' does not exist

Board memory size must be specified to the system using the the host environment variable **IBOARDSIZE**. Details of how to set up **IBOARDSIZE** on your system can be found in the Delivery Manual.

Environment variable 'ITERM' not set up

The **ITERM** definition file for the simulator function keys must be specified in the **ITERM** host environment variable.

IBOARDSIZE is too small (at least *number* bytes required)

The simulator requires a minimum memory size in order to run correctly. Modify **IBOARDSIZE** and retry the command.

ITERM* error*Iterm initialisation has failed**

The *ITERM* file for setting up the terminal codes is invalid. *ITERM* error describes the fault in the file.

Simulator terminated: Error flag set - *message*

Simulator messages may be output when the simulator halts (i.e. as an error condition). *message* can be one of:

- arithmetic overflow
- arithmetic underflow
- long overflow
- subscript out of range
- count out of range
- check single
- check word
- arithmetic exception
- floating point error

Simulator terminated: *message*

Simulator messages may be output when the simulator halts, due to an invalid operation within the program being simulated. *message* can be one of:

attempt made to input from non-existent hard channel

Attempt to input from output link.

attempt made to output to non-existent hard channel

Attempt to output to input link.

attempt to output to unattached hard channel

Attempt to output on unattached link.

attempt to read illegal memory byte at *address*

The memory address specified is invalid (not within **IBOARDSIZE**).

attempt to read illegal memory word at *address*

Invalid memory address or attempt to access non word aligned.

attempt to set illegal memory byte pointer

Invalid memory address (not within **IBOARDSIZE**).

attempt to set illegal memory word pointer

Invalid memory address or attempt to access non word aligned.

attempt to write illegal memory byte at *address*

Invalid memory address (not within **IBOARDSIZE**).

attempt to write illegal memory word at *address*

Invalid memory address or attempt to access non word aligned.

high priority process restored from save area

A swapped out low priority process has been written over during an interrupt.

illegal operand (*nnn*) to operate command

An attempt has been made to execute invalid instruction for the T425.

input from iserver when iserver outputting

ISERVER packet input before leading output sent.

inputting iserver packet larger than expected

Illegal ISERVER protocol packet on input.

output iserver packet larger than expected

Illegal ISERVER protocol packet on output.

output to iserver when iserver inputting

ISERVER packet output before response to last output received.

15 `iskip` - skip loader

This chapter describes the skip loader tool that allows programs to be loaded onto transputer networks over the root transputer. The tool sets up a data transfer protocol on the root transputer that allows programs running on the rest of the network to communicate directly with the host.

15.1 Introduction

The skip tool `iskip` prepares a network to load a program over the root transputer by setting up a transparent route-through process on the root transputer to transfer data from the application program running on the target network to and from the host computer. A subsequent call to `iserver` loads the program onto the network connected to the root transputer, but does not use the root transputer as part of the network. The root transputer is in effect rendered transparent to the rest of the network. The route-through process uses a simple protocol that transfers data byte by byte between the program and the host.

After `iskip` has been invoked to set up the data link across the root transputer, the program can be loaded down the host link using `iserver`. `iskip` can be used to skip any number of processors and load a program into any part of a network, see section 15.2.2.

`iskip` itself may only be executed on 32 bit transputers which have more than 8 Kbytes of memory, although it may be used to reach both 16 and 32 bit transputers for target program execution.

15.1.1 Uses of the skip tool

The skip tool has two main uses :

- 1 To allow programs configured for specific arrangements of transputers to be loaded onto the target network without using the root transputer to run the program. The root transputer helps to load the program onto the network and subsequently provides a route-through process which transfers data from the application program to the host.

Example of boards supplied by INMOS that can be used to skip load programs are the IMS B004 PC add-in board, which contains a single IMS T414 transputer, and the IMS B008 PC motherboard fitted with a TRAM in slot zero to act as the root transputer. Other slots on the motherboard can be used to accommodate the target network.

- 2 Programs configured for a network that normally incorporates the root transputer can be debugged without having to use `idump` to save root

transputer's memory to disk. Programs can be loaded into the network connected to the root transputer and the debugger can safely run on the root transputer without overwriting the program. The external network must have the correct number and arrangement of processors and memory for the program to be loaded.

This can make debugging transputer programs easier when an extra transputer is available.

15.2 Running the skip loader

To invoke the `iskip` tool use the following command line:

► `iskip linknumber { options }`

where: *linknumber* is the link on the root transputer to which the target transputer network is connected.

options is a list, in any order, of one or more options from Table 15.1.

Options must be preceded by '-' for UNIX-based toolsets and '/' for MS-DOS and VMS based toolsets.

Options may be given in any order.

Options may be entered in upper or lower case and can be given in any order.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
E	Directs <code>iskip</code> to monitor the subsystem error status and terminates when it becomes set.
I	Displays detailed progress information as the tool loads.
R	Reset subsystem. Resets all transputers connected downstream of link <i>linknumber</i> . Does <i>not</i> reset the root transputer.
RP	A replacement for the R option when running programs on boards from certain vendors. Contact your supplier to see whether this option is applicable to your hardware. It does not apply to boards manufactured by INMOS.

Table 15.1 `iskip` options

15.2.1 Skipping a single transputer

This example illustrates how to use `iskip` to skip over the root transputer for the example network shown in Figure 15.1.

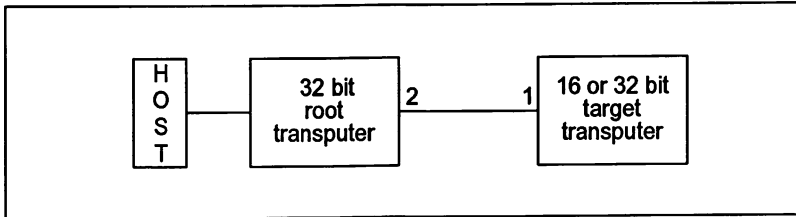


Figure 15.1 Skipping a single transputer

For further information about subsystem wiring see chapter 7 in the *Toolset User Guide* and the debugger documentation in chapter 4 of this manual.

Subsystem wired down:

```

iskip 2 -r                (UNIX based toolsets)
iskip 2 /r                (MS-DOS/VMS based toolsets)
  
```

In this example `iskip` is invoked for a network where the subsystem is wired *down*. The network is prepared to load the program over the root transputer, which is connected to the network via link 2; the '`r`' option resets the target network.

Subsystem wired subs:

```

iskip 2 -r -e            (UNIX based toolsets)
iskip 2 /r /e            (MS-DOS/VMS based toolsets)
  
```

In this example `iskip` is invoked for a network where the subsystem is wired *subs*. The '`e`' option has been added to the example, to direct `iskip` to monitor the subsystem error status, see section 15.2.4.

15.2.2 Skipping multiple transputers

This example illustrates how to use `iskip` to skip over two transputers (starting with the root transputer) for the example network shown in Figure 15.2.

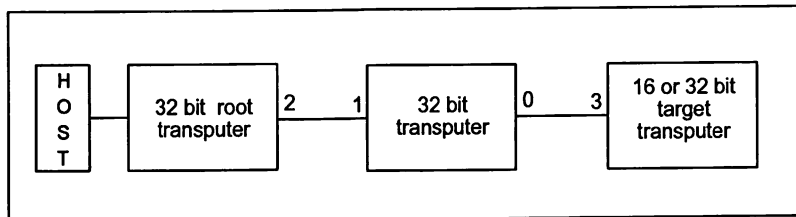


Figure 15.2 Skipping over two transputers

Normally **iskip** is invoked via its driver program; this resets the root transputer and loads the transputer bootable image **iskip.btl** onto the transputer (essentially it performs an **iserver -se -sb iskip.btl** operation). **Note:** because the root transputer is reset, running **iskip** twice in succession will not achieve any more than running **iskip** once; the second invocation will reset the first and load **iskip** onto the root transputer. In order to skip over more than one transputer, **iskip** must be loaded onto subsequent transputers by a 'different' method that does not involve resetting the root transputer. This is best illustrated by an example as shown below (for a network wired *subs*):

```
iskip 2 -r -e                                (UNIX based toolsets)
iserver -se -ss -sc iskip.btl 0

iskip 2 /r /e                                (MS-DOS/VMS based toolsets)
iserver /se /ss /sc iskip.btl 0
```

iskip.btl is the transputer bootable component of **iskip**, it may be found in the **itools** directory of this toolset release. For details of toolset directories see the delivery manual supplied with this toolset.

15.2.3 Loading a program

Once **iskip** has been invoked to prepare the network, the program is loaded by invoking **iserver** with the 'SE', 'SS' and 'SC' options. **iserver** must be invoked with the 'SE' option if the error flag is required to be monitored. This applies whether the **iskip 'E'** option is used or not. For example:

```
iserver -se -ss -sc myprog.btl    (UNIX toolsets)
iserver /se /ss /sc myprog.btl    (MS-DOS/VMS toolsets)
```

Note: After using the skip tool the root transputer must *not* be reset or analyzed, that is, **iserver** must *not* be invoked with the 'SR', 'SB', or 'SA' options, while **iskip** is required to run.

15.2.4 Monitoring the error status – option E

The **iskip 'E'** option should only be used when the sub-network is connected to the Subsystem port of the root transputer i.e. 'wired *subs*'. When the sub-network is connected to the Down port on the root transputer i.e. 'wired *Down*', the 'E' option must not be used.

The 'E' option instructs **iskip** to monitor the subsystem error status and terminate when it becomes set. When it terminates it sets its own error flag in order that the server may detect that an error in the subsystem has occurred. This allows the program to be debugged.

If the subsystem error status is not properly monitored when the program is run, the server may become suspended when a program error occurs. In these circumstances the server can be terminated using the host system BREAK key.

Note: There is a delay of one second after `iskip` is invoked with the 'E' option before monitoring of the subsystem error status begins; if the program fails before this the server may not terminate correctly and the host system BREAK key should be used.

15.2.5 Clearing the error flag

If either `iskip` or `iserver` detect that the error flag is set immediately a program starts executing it is likely that the network consists of more processors than are currently being used and that one or more of the unused processors has its error flag set.

On transputer boards the network may be reset using network check programs, such as `ispy`, which clear all error flags.

The `ispy` program is provided as part of the support software for some INMOS *iq* systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program to clear the network is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared.

15.3 Error messages

This section lists error messages that can be generated by the skip tool.

Called incorrectly

Command line error. Check command line syntax and retry.

Cannot read server's command line

Syntax error. Retry the command.

Duplicate option: *option*

option was supplied more than once on the command line.

No filename supplied

No filename was supplied on the command line.

This option must be followed by a parameter: *option*

The option specified requires a parameter. Check syntax and retry.

Unknown option: *option*

The specified option is invalid. Check option list and retry.

You must specify a link number (0 to 3)

A link number is required. Specify the number of the root transputer link to which the network is connected. If you specify the host link an error is reported.

Appendices

A Toolset conventions and defaults

This appendix describes the standards and conventions used by INMOS toolsets for:

- Command line syntax
- Filenames
- Search paths
- File extensions
- Error message format

A.1 Command line syntax

All tools in the toolsets conform to a common command line format.

A.1.1 General conventions

- Commands, and their parameters and options, obey host system standards.
- Filenames, either directly specified on the command line or as arguments to options, must conform to the host system naming conventions.
- Options must be prefixed with the standard option prefix character for the operating system ('-' for Unix based toolsets and '/' for VMS and MS-DOS based toolsets).
- Command line parameters and options can be specified in any order but must be separated by spaces.
- Lists of arguments to options, where allowed, must be enclosed in parentheses, and the items in the list must be separated by commas.
- If no parameters or options are specified the tool displays a help page that explains the command syntax.

A.1.2 Standard options

Standard command line options used in the toolsets have the same action for all tools. Standard options and their descriptions are given below.

Option	Description
F	Specifies an indirect file (command script).
I	Displays progress data in full.
O	Specifies an output file.

A.2 Unsupported options

A number of tools have various command line options beginning with 'z'. These options are used by INMOS for development purposes and have not been designed for users. As such they are unsupported and should not be used. INMOS cannot guarantee the results obtained from such options nor their continued presence in future toolset releases.

A.3 Filenames

File names generally follow the naming and character set conventions of the host operating system except that the following directory separator characters cannot be used within a filename:

- Colon ':'
- Semi-colon ';'
- Forward slash '/'
- Backslash '\' ('¥' for Japanese MS-DOS)
- Square brackets '[]'
- Round brackets '()'
- Angle brackets '<>'
- Exclamation mark '!'
- Equals sign '='

A.4 Search paths

The tools locate files by searching a specified *directory path* on the host system. The path is specified using the host environment variable **ISEARCH**. The search rules for all tools are as follows:

- 1 If the filename contains a directory specification then the filename is used as given. Relative directory names are treated as relative to the directory in which the tool is invoked.

- 2 If no directory is specified the directory in which the tool is invoked is assumed.
- 3 If the file is not present in the current directory, the path specified by the environment variable (or logical name) `ISEARCH` is searched. If there are several files of the same name on this path, the first occurrence is used.
- 4 If the file is not found using the above rules, then the file is assumed to be absent, and an error is reported.

If no search path has been set up then only rules 1 and 2 apply.

By default all files are written to the current directory.

A.5 Standard file extensions

The INMOS toolsets use a standard set of file extensions for source and object files. In most cases these extensions must be specified on the command line for input files. They are automatically created for output files, unless an alternative file-name is specified on the command line.

A separate set of extensions for object files must be used where `imakef` is used to build programs for mixed processor networks. These are described separately in section A.6.

A.5.1 Main source and object files

Extension	Description
.bt1	Bootable file which can be loaded onto a transputer or transputer network. Created by icollect directly from a .1ku file (single transputer programs) or from a .cfb file. Bootable files can be sent down a link by iserver for immediate execution. Contains information used by iserver to control the host link for execution. Also read by idebug *.
.c	C source files. Assumed by icc , the ANSI C compiler.
.cfb	Configuration binary file containing a description of how code is to be placed on a network, a description of the route to be used to load the network, and the parameters to be passed to each of the processes. Created by the configurer from a user-defined configuration description and read by icollect to prepare a bootable file and by idebug * to load a network for debugging.
.cfs	Configuration description file. This is a text file, created by the user and describes the hardware and software networks and the mapping between them. It also references the linked units and is used as input to the C configurer icconf .
.f77	FORTRAN source files. Assumed by if77 , the FORTRAN-77 compiler.
.h	Header files for use in C source code.
.1ku	Linked unit. Created by ilink as an executable process with no external references. Used as input to icollect (single transputer programs) or within a configuration description. Also read by idebug *.
.lib	Library file containing a collection of binary modules. Created by ilibr .
.occ	Occam source files. Assumed by oc , the OCCAM 2 compiler.
.pgm	Occam configuration description file. This is a text file, created by the user and describes the hardware and software networks and the mapping between them. It also references the linked units and is used as input to the OCCAM configurer occonf .
.tco	Compiled binary module produced by all INMOS TCOFF compilers. Used as input to ilink and ilibr . Also read by idebug *.
* Not applicable to the FORTRAN toolset	

A.5.2 Indirect input files (script files)

Extension	Description
.inc	Include files named in #INCLUDE compiler directives for OCCAM, or #include statements in configuration descriptions or in FORTRAN-77 statements.
.libb	Library build files which specify the components of a library to ilibr .
.liu	Library usage files. Created and used by imakef .
.lnk	Linker indirect files which specify the components of a program to be linked. Also used by imakef when creating Makefiles.

A.5.3 Files read by the memory map tool **imap**

Extension	Description
.mxx	Map file output by the compiler. The characters ' xx ' are determined by the 2nd and 3rd characters of the extension given to the compiler object file. For example if the compiler object file takes the default extension .tco , the information file is given the extension .mco .
.dxx	Map file output by the linker. The characters ' xx ' are determined by the 2nd and 3rd characters of the extension given to the linker object file. For example if the linker object file takes the default extension .1ku , the information file is given the extension .dku .
.map	Map file output by the collector.
Note: These extensions also satisfy imakef 's requirements, see section A.6.	

A.5.4 Other output files

Extension	Description
.bin	Binary format files produced by ieprom for loading into ROM.
.btr	Executable code without a bootstrap. Created by icollect and used as input to ieprom .
.clu	Configuration object file, created by the OCCAM configurer occonf .
.hex	A hex dump of a file for loading onto a ROM by a custom ROM loader tool.
.ihx	Intel hex format files produced by ieprom for loading into ROM.
.mot	Motorola 'srecord' files produced by ieprom for loading into ROM.
.rsc	An .rsc file contains the code of a process together with a description of its requirements for data areas and parameters. It is created by the collector from a linked unit. The format is described in chapter 3. .rsc files are suitable for using with either the OCCAM or C functions which support dynamic code loading.

A.5.5 Miscellaneous files

Extension	Description
.dmp*	Memory dump and network dump files. Created by idump for debugging code on the root transputer (memory dump) or by idebug for off-line analysis of a program on a network (network dump). Read by the debugger for post-mortem debugging.
.itm	ITRM files containing information about the terminal. Used by tools such as idebug to handle the screen in a device-independent manner. Can also be created by users for other terminals. The file is referenced via the environment variable ITRM .
.mak	Makefile generated by imakef . This file may be input to a "make" utility to build the target file. May also be edited by the user.
* Not applicable to the FORTRAN toolset	

A.6 Extensions required for **imakef**

The standard set of file extensions are adequate for simple programs executing on a single transputer, or on a network of transputers all of the same type. If the network is heterogeneous and a particular source file needs to be compiled for more than one transputer type, the following scheme can be used to identify the individual processor types and error modes.

If **imakef** is used to build the program, this scheme *must* be used.

The extended system uses extensions of the form **.fpe**:

where: *f* denotes the type of file and can take the following values:

t for **.tco** equivalents.

l for **.lnk** equivalents.

c for **.lku** equivalents.

r for **.rsc** equivalents.

p denotes the transputer target type or class. This can take the following values:

2 – T212, T222, M212

3 – T225

4 – T414

5 – T425, T400, T426

8 – T800

9 – T801, T805

a – T400, T414, T425, T800, T801, T805

b – T400, T414, T425

e denotes the execution error mode. The values it can take are:

h – on execution, an error will immediately halt the transputer.

s – when an error occurs, the transputer's error flag will be set.

x – the program can be executed in either HALT or STOP mode.

A.7 Message handling

All tools in the toolsets display diagnostic messages in a standard format. This has certain advantages:

- 1 The tool generating the message can be identified even when the tool is run out of contact with the terminal.
- 2 User programs or system utilities can be used to detect and manipulate errors. Some host system editors permit automatic location of errors.

A.7.1 Message format

Diagnostic messages are displayed in a standard format by all tools. The generalized format can be expressed as follows:

severity – *toolname* – *filename* (*linenumber*) – *message*

where: *severity* indicates the severity level. Severity categories are described below.

toolname is the standard toolset name for the tool. Names defined using host system abbreviations and batch files are not displayed.

filename and *linenumber* indicate the file and line where an error occurred. They are only displayed if the error occurs in a file. They are commonly displayed when files of the wrong format are specified on the command line, for example, a source file is specified where an object file is expected.

message explains the error and may recommend an action.

A.7.2 Severities

The severity attached to the message indicates the importance of the diagnostic to the operation of the tool. It also implies a certain action taken by the tool.

Five severity categories are recognized:

Information Warning Error Serious Fatal

Information messages provide the user with information about the functioning or performance of the tool. They do not indicate an error and no user action is required in response.

Warning messages identify minor logical inconsistencies in code, or warn of the impending generation of more serious errors. The tool continues to run and may produce usable output if no serious errors are encountered subsequently.

Error messages indicate errors from which the tool can recover in the short-term but may cause further errors to be generated which may lead to termination. The tool may continue to run but further errors are likely and the tool is likely to abort eventually. No output is produced.

Serious errors are errors from which no recovery is possible. Further processing is abandoned and the tool aborts immediately. No output is produced.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your local INMOS distributor or field applications engineer.

A.7.3 Runtime errors

Errors which prevent the program from being run are detected by the runtime system at startup or during program execution. These errors are displayed in a similar format to that used by the tools. All runtime errors are generated at *Fatal* severity and cause immediate termination of the program.

B Transputer types and classes

This appendix first identifies the INMOS transputer types supported by this toolset. It then explains the concept of transputer classes in terms of developing programs for multiple transputer targets. This includes compiling and linking program modules. The examples given are based on the 'Hello world' program, written in C and compiled with the `icc` compiler.

It also explains the command line options which can be used to specify a target processor or transputer class.

Note: the information given in this appendix covers the current range of INMOS transputers and language compilers; readers should ignore details of transputer types or languages which do not apply to this particular toolset.

B.1 Transputer types supported by this toolset

The ANSI C and occam 2 toolsets can be used to develop programs targetted at the following INMOS transputer types:

IMS M212, T212, T222, T225, T400, T414, T425, T426, T800, T801, T805.

The FORTRAN toolset can only be used to develop programs targetted at 32-bit transputers. This includes the following INMOS transputer types:

IMS T400, T414, T425, T426, T800, T801, T805.

The default type assumed by various tools, if none is specified on the command line, is T414.

B.2 Transputer types and classes

This section describes the meaning of transputer types and classes and how selection of the target processor affects the compilation and linking stages of program development. The section describes how to compile and link code targeted at a single processor type and then describes how to compile and link programs so that they can be executed on different processor types.

B.2.1 Single transputer type

For those who have a single transputer or indeed a network of transputers all of the same type, the compilation and linking stages of program development are very straightforward. Simply compile and link all your modules for the required processor.

Example: to compile and link for a T800:

```
icc hello -t800
ilink hello.tco -t800 -f cstartup.lnk (UNIX)

icc hello /t800
ilink hello.tco /t800 /f cstartup.lnk (MS-DOS and VMS)
```

For a T414 the command lines are simpler:

```
icc hello
ilink hello.tco -f cstartup.lnk (UNIX)

icc hello
ilink hello.tco /f cstartup.lnk (MS-DOS and VMS)
```

B.2.2 Creating a program which can run on a range of transputers

The compiler and linker use the concept of transputer *class* to enable programs to be developed which may be run on different transputer types without the need to recompile.

A transputer class identifies an instruction set which is common to all the processors in that class. When a program is compiled and linked for a transputer class it may be run on any member of that class.

Note: Code created for a transputer class will often be less efficient than code created for a specific processor type. Therefore, creating code for a transputer class is discouraged in situations where program efficiency is a primary concern; it should only be performed where there is a genuine need to produce code which will run on a range of transputers or to reduce the size of a support library, where program efficiency is not a major concern.

Table B.1 lists all the transputer classes which the compiler and linker support and indicates which processors the program can be run on.

Transputer	Processors which class can be run on
T2 *	T212, M212, T222, T225
T3 *	T225
T4	T414, T400, T425, T426
T5	T400, T425, T426
T8	T800, T801, T805
T9	T801, T805
TA	T400, T414, T425, T426, T800, T801, T805
TB	T400, T414, T425, T426
* Not applicable to the FORTRAN toolset	

Table B.1 Transputer classes and target processor

In order to develop a program which will run on different processor types, perform the following steps:

- 1 Identify the processors on which the program is to run.
- 2 Using Table B.1 select the class which may be run on all the target processors.
- 3 Compile and link all the program modules for this class.

For example, to create a program which will run on both a T400 and a T425, compile and link for transputer class T5:

```
icc hello -t5
ilink hello.tco -t5 -f cstartup.lnk    (UNIX)
```

```
icc hello /t5
ilink hello.tco /t5 /f cstartup.lnk    (MS-DOS and VMS)
```

Alternatively to create a program which will run on a T400, T425 or a T800, compile and link for transputer class TA:

```
icc hello -ta
ilink hello.tco -ta -f cstartup.lnk    (UNIX)
```

```
icc hello /ta
ilink hello.tco /ta /f cstartup.lnk    (MS-DOS and VMS)
```

Code compiled for a T414 (class T4) may be run on a T400 or T425, which form class T5.

Programs compiled for the T212, M212, or T222 transputers i.e. class T2, can be run on a T225 (class T3) because a T225 has a similar but larger instruction set than class T2 transputers. Similarly the T400, T425 and T426 have additional instructions to those of the T414. Likewise, code compiled for a T800 (class T8) may be run on a T801 or T805, which form class T9. Again the T801 and T805 have additional instructions to those of the T800. See section B.2.4.

B.2.3 Linked file containing code compiled for different targets

This section describes how object code compiled for one target processor or transputer class can be linked with code compiled for different transputer types or classes.

The ability to do this provides the user with greater flexibility in the use of program modules:

- An individual module can be compiled once e.g. for class T4, and then linked with separate programs to run on different processor types e.g. T414 and T425.

- When the user is preparing a library for use by programs intended to run on different processor types, a single copy of code compiled for a transputer class can be inserted instead of multiple copies for specific transputers.

When linking a collection of compiled units together into a single linked unit, the user must select a specific transputer type or transputer class on which the linked unit is to run. As before, this determines the set of transputer types on which the code will run. When linking for a particular type or class, the linker will accept compilation units compiled for a compatible class. Table B.2 shows which transputer types and classes the linker will accept when linking for a particular class.

Link class	Transputer classes which may be linked
T2 *	T2
T3 *	T3, T2
T4	T4, TB, TA
T5	T5, T4, TB, TA
T8	T8
T9	T9, T8
TB	TB, TA
TA	TA
* Not applicable to the FORTRAN toolset	

Table B.2 Linking transputer classes

For example, if the target processors are a T400 and a T425 the user may compile for classes T5 and TB and link the code for for class T5. Code for a different transputer class can be included in the final linked unit, as long as:

- It uses the instruction set or a subset, of the instruction set of the link class
- The calling conventions are the same.

Classes T8 and T9 cannot be linked with class TA. This is a change from early toolset releases e.g. the Dx11 C toolsets, the Dx05 occam toolsets and the Dx13 3L FORTRAN toolsets.

The reason why these classes cannot be linked together is explained in section B.2.4. which gives details of the differences between the instruction sets, as additional information.

A library can be made, consisting of the same modules compiled for different transputer types or classes. The user then needs only to specify the library file to the

linker, and the linker will choose a version of a required routine which is suitable for the system being linked.

The linker uses the rules given in Table B.2 to determine whether a compiled module, found in a library, is suitable for linking with the current system. So, for example, to create a library which may be linked with any transputer class or specific transputer type, all routines could be compiled for classes T2, TA and T8.

If there are a number of possible versions of a module in a library the best one (i.e. the most specific for the system being linked) is chosen.

occam object files targetted at different targets

For OCCAM programs the above rules must also be applied during the program design stage when deciding which modules should call each other. Code for a different transputer class can be called provided that it uses the instruction set or a subset of the instruction set of the calling class. (This is because the compiler needs to know which modules to select from libraries containing copies for different processor types).

Table B.2 can be used as guide, by regarding the 'link class' as the 'calling class' and the 'transputer classes which may be linked', as the 'transputer classes which may be called'.

Note: classes T8 and T9 cannot call class TA.

B.2.4 Classes/instruction sets – additional information

The instruction sets of the transputer classes differ in the following ways:

- Classes T2 and T3 support 16-bit transputers whereas all the other transputer classes support 32-bit transputers.
- Class T3 is the same as class T2 except that T3 has some extra instructions to perform CRC and bit operations and includes special debugging functions.
- Class T5 is the same as class T4 except that T5 has extra instructions to perform CRC, 2D block moves, bit operations, special debugging functions and also includes the *dup* instruction.
- Class T9 is the same as class T8 except T9 has additional debugging instructions.
- The T800, T801 and T805 processors use an on-chip floating point processor to perform REAL arithmetic. Thus a large number of floating point instructions are available for these transputers and for their associated classes T8 and T9. These instructions are listed in the instruction set appendix of the *User Guide*.

- For the T414, T400, T425 and T426 processors i.e. transputer classes T4 and T5 the implementation of REAL arithmetic is in the software. These transputers make use of a small number of floating point support instructions. Details can be found in the instruction set appendix of the *User Guide*.
- The instruction set of class TA only uses instructions which are common to the T400, T414, T425, T426, T800, T801 and T805 transputers. Therefore it does not use the floating point instructions, the floating point support instructions or the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.
- The instruction set of class TB only uses instructions which are common to the T400, T414, T425 and T426 processors. Therefore it uses the floating point support instructions, but does not use the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.

When considering the similarities and differences in the instruction sets of different transputer classes it helps to divide them into the separate structures as shown in Figure B.1.

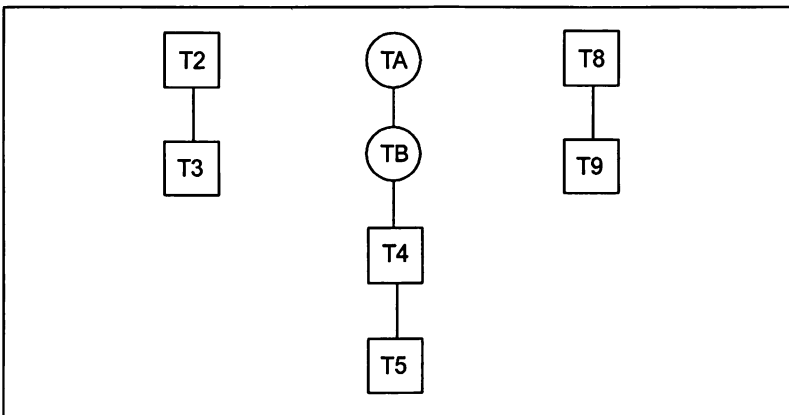


Figure B.1 Structures for mixing transputer types and classes

By comparison with Table B.2 it can be seen that a module may only be linked with modules compiled for a transputer class which belongs to the same structure.

Classes T2 and T3 are targetted at 16-bit transputers so it is obvious that they cannot be linked with the other classes which are all targetted at 32-bit transputers.

The reason why classes T8 and T9 cannot be linked with classes TA, TB, T5 or T4 is because floating point results from functions are returned in a floating point

register for T8 and T9 code and in an integer register for all other 32-bit processors. Even if your code does not perform real arithmetic, linking code compiled for a T9 or T8 with code compiled for any of the other classes is not permitted.

To summarize, compiling code for the transputer classes TA and TB enables it to be run on a large number of transputer types, however, the code may not be as efficient as code compiled for one of the other transputer classes or for a specific transputer type. For example compiling code for class T5 enables the CRC and 2D block move instructions to be used, whereas these instructions are not available to code compiled for classes TA and TB.

B.3 Transputer type command line options

This section lists the command line options used to specify a target processor or transputer class. The options can be used with the following tools:

- `icc` - The ANSI C compiler.
- `oc` - The OCCam 2 compiler.
- `if77` - The FORTRAN-77 compiler.
- `ilink` - The toolset linker.

Option	Description
TA	Specifies target transputer class TA (T400, T414, T425, T426, T800, T801, T805).
TB	Specifies target transputer class TB (T400, T414, T425, T426)
T212	Specifies a T212 target processor.
T222	Specifies a T222 target processor. Same as T212
M212	Specifies a M212 target processor. Same as T212
T2	Same as T212, T222 and M212
T225	Specifies a T225 target processor.
T3	Same as T225.
T400	Specifies a T400 target processor. Same as T425.
T414	Specifies a T414 target processor. This is the default processor type and may be omitted when the target processor is a T414 processor.
T4	Same as T414 (default).
T425	Specifies a T425 target processor.
T426	Specifies a T426 target processor.
T5	Same as T400, T425 and T426.
T800	Specifies a T800 target processor.
T8	Same as T800.
T801	Specifies a T801 target processor. Same as T805.
T805	Specifies a T805 target processor.
T9	Same as T801 and T805.

Table B.3 Transputer type command line options

C Using the assembler

This appendix describes the assembler supplied with the ANSI C toolset. The appendix explains how to invoke the assembler and describes the use and syntax of assembly directives. The chapter ends with a list of assembler diagnostic messages which may be obtained.

C.1 Introduction

The assembler is supplied as an integral part of the ANSI C compiler `icc` and is normally run as the final stage of compilation. The command line interface of the compiler enables the user to invoke the assembler directly. This suppresses the compilation phase of the compiler and the input file is passed directly to the assembler.

The assembler is a cross-assembler which enables a source file written in assembly code to be translated into object code. The assembler accepts as input a single ASCII text file consisting of transputer instructions and assembler directives.

If required the compiler preprocessor can be run on an assembler source file and the output file generated by the preprocessor used as input to the assembler.

The assembler generates an object file in Transputer Common Object File Format (TCOFF). This file may then be linked with other TCOFF object files or library modules, configured, loaded onto a transputer network and the application executed.

C.2 Running the assembler

The assembler is invoked by using the `'AS'` option on the ANSI C compiler `icc`.

The assembler is invoked to assemble a source file by one of the following command lines:

<code>icc filename.ext -as {options}</code>	(UNIX)
<code>icc filename.ext /as {options}</code>	(MS-DOS/VMS)

where: *filename.ext* is the filename and extension of the assembler source file, see section C.2.1.

options is a list of `icc` command line options, see section C.2.2.

A list of error messages which may be generated by the assembler is given in section C.6.

C.2.1 Specifying the source filename

The full filename including extension must always be supplied on the compiler command line when the `'AS'` option is used. If an extension is not supplied the com-

pilc assumes a C source file is to be compiled and searches for an input file which has the appropriate name and '.c' file extension.

C.2.2 Use of icc command options with the assembler

Many of the `icc` command line options have no meaning if used with the assembler and will be ignored. The only options which have meaning are:

- Any option to select the target transputer type e.g. T805, T400 (see appendix B).
- 'I' – Displays progress information as the tool runs.
- 'O filename' – Specifies an output filename.

`icc` command line options are documented in full in chapter 1.

C.2.3 Using the pre-processor with the assembler

Preprocessor directives may be included in assembler source files, pragma directives, however, should not.

When preprocessor directives are used, the compiler preprocessor must be run on an assembler source file prior to using the assembler. The output file generated by the preprocessor is then used as input to the assembler. The preprocessor is invoked using the compiler command line option 'PP'.

For example:

UNIX based toolsets:

```
icc test.s -pp > temp.s
icc temp.s -as -o test.tco
```

MS-DOS based toolsets:

```
icc test.s /pp > temp.s
icc temp.s /as /o test.tco
```

VMS based toolsets:

```
DEFINE SYS$OUTPUT temp.s
icc test.s /pp
DEASSIGN SYS$OUTPUT
icc temp.s /as /o test.tco
```

The first time `icc` is run, the 'pp' option is used to invoke the preprocessor. The preprocessor sends output to `stdout` by default. In the UNIX and MS-DOS versions of the example, output is redirected by the redirection operator '>' to a temporary output file. In the VMS version, system output is defined to be a temporary file. (It is reset later by using the `$DEASSIGN` command).

The temporary file is input on the compiler command line and the assembler is run. The 'o' is used to specify an output file for the assembled code.

C.3 Language

An assembler source file is made up of lines of ASCII text. Each line can contain the following:

- A label definition.
- Assembler commands separated by semi-colons or new lines.
- A comment.

None of the above may extend to more than one line.

An assembler command is one of the following:

- A transputer instruction mnemonic (with its operand, if any)
- An assembler directive.

C.3.1 Label definitions

Label names can contain alphanumeric characters and the characters '.' (full-stop) and '_' (underscore). A label is defined by terminating its name by a colon:

```
label:
```

A label is used by giving its name without the colon, e.g.

```
j label          -- jump to label
```

Labels are also referred to as code symbols.

C.3.2 Symbols

There are three kinds of symbols in the assembler. They are as follows:

- Code symbols : These are labels as defined above.
- Data symbols : These are symbols introduced by the `data` or `common` directives.
- Defined symbols : These are symbols defined using the `defsymb` directive.

Once a symbol has been defined as one kind of symbol it cannot be redefined to another.

C.3.3 Expressions

The assembler can recognize simple integer expressions constructed from operators, operands and parentheses. An operator performs a mathematical or logical

operation on the operand(s) in the expression. Allowable operators are listed in table C.1.

Operator	Meaning	Precedence
-	unary minus	1
~	bitwise not	1
*	multiplication	2
/	division	2
%	modulus	2
+	addition	3
-	subtraction	3
&	bitwise and	4
	bitwise or	4
^	bitwise exclusive or	4
<	logical left shift	5
>	logical right shift	5

Table C.1 Operators

Evaluation is left-to-right, except for unary operations where the operator closest to the operand binds more tightly, with precedence rules as follows. The lower the number in the precedence column of the table, the higher the precedence of the operator. Parentheses, (...), may be used to alter the order of evaluation of an expression.

All calculations are done in 32 bits regardless of the word size of the target machine. Overflow in expression generation in the assembler is avoided by performing addition, subtraction and multiplication as modulo operations. Division and modulus by zero will result in a serious error being reported. The special case of `MOSTNEG_INT / -1` also results in a serious error message.

Table C.2 indicates the different types of operand which may appear within expressions:

Operand	Value
Number	Its numerical value.
Defined symbol	The value assigned to the symbol using <code>defsym</code> .
Code symbol	The offset in bytes from the end of the instruction containing the expression to the symbol.
Data symbol	The word offset of this symbol in the data area.

Table C.2 Operands

Note: code and data symbols may not appear in the same expression and they may only appear in expressions which are operands to primary transputer instructions, as defined in appendix A of the *ANSI C Toolset User Guide*.

C.3.4 Transputer instruction mnemonics

The transputer instructions supported by the assembler are listed in appendix A of the *ANSI C Toolset User Guide*; detailed information is given in the '*Transputer instruction set – a compiler writer's guide*'. **Note:** no check is made that the transputer instructions used in the source code are supported by the transputer target selected on the compiler command line.

C.3.5 Comments

A comment is introduced by the characters `—`. If a comment needs to be split over more than one line, each line must start with the characters `—`. All text appearing on the same line and to the right of these characters is interpreted as a comment. For example:

```
— A comment on a line of its own
  j fred — A comment at the end of a line
```

C.4 Assembler directives

This section briefly describes each assembler directive and provides an example where appropriate. Directives appear in alphabetical order.

Table C.3 summarizes the directives available.

Directive	Description
align	Aligns byte to word boundary.
blkb	Generates a block of bytes.
blkw	Generates a block of words.
byte	Generates a sequence of bytes set to specified values.
comment	Causes a comment to be written to the object file.
common	Defines a FORTRAN common block.
data	Defines a symbol to be a data symbol.
debug	Generates a debug information record.
defsym	Assigns a value to a symbol. Used only for local symbols within an assembler file.
descriptor	Creates an occam style descriptor in the object file.
extern	Declares a symbol to be external to the module.
global	Defines a symbol to be globally visible.
init	Defines a member of the static initialization chain.
language	Defines the language of the current module.
maininit	Used to find the start of the static initialization chain.
map1	Generates text information for a memory map file.
map2	Generates symbol information for a memory map file.
patch	Generates a patch. Six patch types are available.
size	Pads out an instruction so that it occupies a specified number of bytes.
sourcefile	Overrides the default source file name with the specified name.
textname	Replaces the default code section name for the current module with the specified name.
toolname	Overrides the record of the tool used to create the object file, with the specified tool name.
word	Generates a sequence of words set to specified values.

Table C.3 Assembler directives

The names of assembler directives are reserved keywords.

align

Syntax:

align

Description:

The **align** directive causes the next generated byte in the code section to be aligned on the next word boundary as defined by the target word length of the processor.

blkb

Syntax:

```
blkb <expr> [, <expr_or_string>]
```

Description:

The **blkb** directive generates a block of bytes. The number of bytes to be generated is given by the first expression, the **size**. The value of the bytes is given by the operands which follow the first expression.

If no operands are given then **size** zero bytes are generated.

If the operand is an expression and its value is too large to fit in a byte then an error is reported. See section C.6.

If the operand is a string then the characters of the string are written to consecutive bytes of memory.

If the length of the string exceeds the specified size then the trailing bytes are ignored.

If too few expressions are given then the remaining bytes are set to zero. It is an error to give too many expressions to the **blkb** directive.

Example:

```
blkb 10, "hello", 6, 7, 8, 9, 10
```

This generates the byte values 'h', 'e', 'l', 'l', 'o', 6, 7, 8, 9, 10 to consecutive bytes of memory.

blkw

Syntax:

```
blkw <expr> [, <expr>]
```

Description:

The **blkw** directive generates a block of words. The number of words to be generated is given by the first expression, the **size**. The value of the words is given by the operands which follow the first expression.

If no operands are given then **size** zero words are generated. This result is also obtained if the **size** given is '0' or a negative quantity.

If too few expressions are given then the remaining words are set to zero. It is an error to give too many expressions to the **blkw** directive. Each word is stored in little-endian format.

Example:

```
blkw 3 + 4, 1, 2, 3, 4, 5, 6, 7
```

The above generates 7 words in memory with the values 1, 2, 3, 4, 5, 6 and 7.

byte

Syntax:

```
byte <expr_or_string> [, <expr_or_string>]
```

Description:

The **byte** directive generates a sequence of bytes each of which takes on the values of the following operands.

If the operand is an expression and the result of the expression is too large to fit in a byte then an error is generated. See section C.6.

If the operand is a string then the characters of the string are assigned to consecutive bytes of memory.

Example:

```
byte "hello", 6, 7, 8, 9, 10
```

This generates the byte values 'h', 'e', 'l', 'l', 'o', 6, 7, 8, 9, 10 to consecutive bytes of memory.

comment

Syntax:

```
comment <string>
```

Description:

This directive causes the string to be written to the object file as a TCOFF comment. The comment is printable but cannot be copied so it will not appear in a linked unit.

A comment can be seen by using the `lister` tool on an object file. The `ilist` command line option `'m'` is used.

Example:

To write the comment "Hello" to the object file:

```
comment "Hello"
```

common

Syntax:

common <symbol> <expr>

Description:

The **common** directive defines a FORTRAN common block denoted by the symbol **symbol**, with size in words given by the expression. A common block resides in a TCOFF section of its own.

The final size of a common block is given by the **common** directive for a given symbol with the greatest size which is present in the link.

This directive is included for completeness, it has no application in C programs.

data

Syntax:

data <symbol> <expr>

Description:

The **data** directive defines **symbol** as a data symbol. The size in words of the data item is given by the expression and this much space is reserved in the current module's data area. The space is allocated from above the last data symbol or from the start of the area if there were no previous data symbols. The value of a data symbol is its word offset into the current module's data area.

Example:

To define a data symbol **fred**, 1 word long and to make it visible outside of this module:

```
data fred 1    -- define the data symbol, 1 word long
global fred    -- make it global
```

debug

The **debug** directive defines a debug information record and is designed for use by compilers and other programs generating assembly language output. It is listed here for completeness.

defsym

Syntax:

```
defsym <symbol> <expr>
```

Description:

This directive allows a numeric value to be assigned to the **symbol**. The value is given by the expression. This symbol can then be used in expressions as if it were the value itself.

This is only for use internally to an assembler file. These symbols cannot be made global and used in other files. A symbol which is defined using a **defsym** directive must not have been previously defined as a code or data symbol and vice versa.

Code and data symbols may not appear in the expression, only symbols defined using **defsym** are permissible. Any symbol used in the expression must have been previously defined.

Example:

To set the symbol 'one' to be the value 1:

```
defsym one 1
```

one can now be used in expressions as if it were the value 1.

descriptor

Syntax:

```
descriptor <symbol> <string> <language_type> <expr> <expr> <string>
```

Description:

This directive causes an OCCAM style descriptor to be output to the object file. It also causes two symbols to be defined which are used by some other tools, including the toolset collector, to obtain workspace and vector space information.

The **symbol** is a code symbol denoting the routine for which the descriptor is to be created.

The first string is used as a prefix for two symbols which contain the workspace and vector space requirements, (in words) for the routine.

language_type is a language type of the same form as that for the **language** directive. The convention is that the language type denotes the source language used in the interface description contained in the descriptor string, however a language type must still be supplied even if the descriptor string is an empty string.

The next two expressions are the workspace and vector space requirements respectively and the last string is the descriptor string itself.

Example:

To define an OCCAM descriptor for the routine **FRED** which requires 42 words of workspace, no vector space and has the following definition:

```
PROC FRED([18]INT ProcessData)
  SEQ
  ... lots of work
:
```

use the following directive:

```
FRED:
  global FRED
  descriptor fred "FRED" occam 42 0 "PROC FRED([18]INT ProcessData)\n SEQ\n:"
```

This generates the following TCOFF records:

```
00000053 SYMBOL EXP UNI "FRED'ws" id: 3
0000005E SYMBOL EXP UNI "FRED'vs" id: 4
00000069 DEFINE_SYMBOL id: 3 42
0000006E DEFINE_SYMBOL id: 4 0
00000073 DESCRIPTOR id: 2 lang: OCCAM
ws: 42 vs: 0
PROC FRED([18]INT ProcessData)
  SEQ
:
```

TCOFF records can be listed with **ilist**, using the command line 't' option.

extern

Syntax:

extern <symbol>

Description:

The **extern** directive is used to declare the symbol **symbol** as external to the current module. An example of its use is to call an external routine (see the description of the **patch** directive for an actual example).

global

Syntax:

global <symbol>

Description:

The **global** directive is used to cause the symbol to become available outside of the current module. It causes the symbol to become globally visible so that it can be accessed by the linker.

The operand to a **global** directive i.e. the symbol, may be a code or data symbol.

Note: it *must not* be a symbol defined by the **defsym** directive. The **global** directive can appear before the definition of its operand.

Example:

To create a routine called **fred** which can be called from external modules:

```
fred:
    global fred
    -- do freds operations
```

init

Syntax:

`init`

Description:

The `init` directive is used to define a member of the static initialization chain. The static initialization chain is a list of routines which are called by the runtime initialization code, in order to set up the static area.

Each routine in the list is introduced by the `init` directive, which defines the location of a word in memory which is used to link the members of the chain together. After linking, this word holds the byte offset to the next `init` word or zero if it is the end of the chain. The byte directly following the `init` word is the first executable byte of the initialization routine.

Note: that `init` does not itself reserve the word in memory. It is necessary to follow the `init` directive immediately with a directive or dummy instructions to reserve the space. INMOS initialization routines are called with one parameter, a pointer to the start of the static area.

Example:

For a 32 bit machine:

```
align
init                                -- init directive
byte    #20, #20, #20, #20         -- space reserved for init word
ldc 1                                -- first executable instruction of
                                -- initialization routine.

ldc 2
ret                                -- end of initialization routine
```

language

Syntax:

`language <language_type>`

Description:

Sets the language for the current module. `language_type` is one of the following:

- `unknown`
- `occam`
- `ansi_c`
- `fortran`
- `iso_pascal`
- `modula2`
- `ada`
- `assembler`
- `occam_harness`

If no `language` directive appears in the input file then the language defaults to `assembler`. The language type is written into the TCOFF `START_MODULE` record.

`occam_harness` is a special language type used to define language runtime system main entry points for use by the configurer.

Example:

To set the language type for the current file to be `ada`:

```
language ada
```

maininit

Syntax:

maininit

Description:

The **maininit** directive is used to find the start of the static initialization chain. The **maininit** directive defines the location of a word in memory into which the linker will patch the byte offset to the first routine in the static initialization chain. In other words, after linking, the word defined by the **maininit** patch contains the byte offset to the location of the first **init** word in the static initialization chain.

Note: that **maininit** does not itself reserve the word in memory. It is necessary to follow the **maininit** directive immediately with a directive or dummy instructions to reserve the space.

Note: In order to use a **maininit** directive, an **init** directive must be present somewhere in the link. If this is not the case then the link will fail.

Example:

To obtain the start of the initialization chain on a 32 bit machine:

```

        align
.mainlab:                                -- label so we can find this
                                           -- word
        maininit                         -- maininit directive
        byte    #20, #20, #20, #20      -- space reserved for maininit
                                           -- word
        ldc (.mainlab - .label)          -- load the address of the
                                           -- maininit word
        ldpi                             -- load a pointer to this
                                           -- address
.label:
        ldnl 0                          -- load the contents of the
                                           -- maininit word
                                           -- into the A register

```

map1

Description:

This directive is used internally by the compiler to generate text information for a map file. It is listed here for completeness and is not intended for use in customers' assembly source files.

map2

Description:

This directive is used internally by the compiler to generate symbol information for a map file. It is listed here for completeness and is not intended for use in customers' assembly source files.

patch

Overview:

There are six different types of `patch` directive. They are:

- **CODEFIX**
- **DATAFIX**
- **EXTOFFSET**
- **LIMIT**
- **MODNUMBER**
- **STATICFIX**

Each is discussed in detail below. **Note:** that no space is reserved by the `patch` directive. The appropriate number of bytes should be reserved following the `patch` directive using other directives or dummy instructions.

Each of the six types of patch can come in three forms, they are:

- **Instruction.** Denoted by the patch directive containing a primary instruction mnemonic. In this case the value of the patch becomes the operand to the instruction and this instruction/operand combination is patched into the hole in the code. If the instruction/operand combination is shorter than the number of bytes reserved in the `patch` directive then it occupies the start of the reserved space and the unused trailing bytes are filled with `pfix 0` instructions (hex 20).
- **Short.** Denoted by the patch directive containing the word `short`. In this case the value of the patch is patched directly into the hole in the code reserved for it. The value patched will occupy 2 bytes irrespective of the target processor word length and the hole reserved must reflect this.
- **Long.** Denoted by the patch directive containing the word `long`. In this case the value of the patch is patched directly into the hole in the code reserved for it. The value patched will occupy 4 bytes irrespective of the target processor word length and the hole reserved must reflect this.

The first operand of a patch directive is the patch size in bytes. The size of a patch must be between one and 255 bytes inclusive. An attempt to issue a patch with a size outside this range will result in an error being reported. Also the size given for a `short` patch must be two bytes and the size given for a `long` patch must be four bytes otherwise an error is reported.

patch - codefix

Syntax:

```
patch <expr> <instruction> codefix <symbol> <expr>
```

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the offset between the byte following the patched instruction or value and the address of the symbol *symbol*. In the case where the patch is an instruction patch and the patched instruction does not entirely fill the reserved space, the value of the instruction operand is the offset between the first unused trailing byte and the address of the symbol. The *symbol* must be a code symbol.

The final expression is an offset which is added to the value of the patch. *instruction* is a transputer primary instruction or the tokens *short* or *long* (see the overview of the *patch* directive for an explanation of these).

Example:

To call an external function

```
extern  _IMS_printf          -- declare an external symbol
patch   6 j codefix _IMS_printf 0 -- do the patch
byte    #20, #20, #20, #20, #20, #20 -- hole for the patch
```

In the above example the patch is six bytes long. The patch produces a *j* (jump) instruction, the operand of which is the offset between the instruction after the patch and the symbol *_IMS_printf*. Therefore the jump instruction will transfer control to the instruction at the address of the *_IMS_printf* symbol. The offset zero in this example, causes a jump directly to the symbol.

patch - datafix

Syntax:

```
patch <expr> <instruction> datafix <symbol> <expr>
```

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the offset, in words, between the start of the local static area for this module and the symbol *symbol*, plus the value of the second expression, the offset.

The offset can be used to access elements of structures etc. The symbol must be a data symbol. *instruction* is a transputer primary instruction or the tokens *short* or *long* (see the overview of the *patch* directive for an explanation of these).

Example:

To obtain the offset of *fred* from the start of the local static area for this module:

```
extern fred
patch 4 long datafix fred 0
byte #20, #20, #20, #20
```

The above example patches the offset, in words, from the start of the local static area to the location of *fred*, into the four byte hole.

patch - extoffset

patch <expr> <instruction> **extoffset** <symbol> <expr>

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the offset to the symbol *symbol* from the start of the section containing it. If the symbol is a code symbol this offset is in bytes. If the offset is a data symbol the offset is in words. The final expression is an offset which is added to the value of the patch.

instruction is a transputer primary instruction or the tokens *short* or *long* (see the overview of the *patch* directive for an explanation of these).

Example:

To obtain the byte offset to *main* from the start of the text section:

```
extern main                -- declare an external symbol
symbol
  patch 4 long extoffset main 0  -- do the patch
  byte #20, #20, #20, #20      -- hole for the patch
```

In the above example the patch is 4 bytes long. The offset of *main* in the text section is patched into the 4 byte hole.

patch - limit

```
patch <expr> <instruction> limit
```

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the size, in words, of the global static area for the entire link. **Note:** that this includes any common blocks which are defined.

instruction is a transputer primary instruction or the tokens **short** or **long** (see the overview of the **patch** directive for an explanation of these).

Example:

To obtain the static size for the program:

```
patch 6 ldc limit  
byte #20, #20, #20, #20, #20, #20
```

The above example patches a **ldc** instruction into the 6 byte hole. The operand of the instruction is the size of static used by the program in words.

patch - modnumber

```
patch <expr> <instruction> modnumber
```

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the current module number. Each module is identified by a unique module number within a link.

instruction is a transputer primary instruction or the tokens *short* or *long* (see the overview of the *patch* directive for an explanation of these).

Example:

To obtain the module number for the current module:

```
patch 6 ldc modnumber  
byte #20, #20, #20, #20, #20, #20
```

The above example patches a *ldc* instruction into the 6 byte hole. The operand of the instruction is the current module number.

patch - staticfix

patch <expr> <instruction> staticfix

Description:

This creates a patch of size *n* bytes, where *n* is given by the first expression in the directive. The value of this patch is given by the offset between the start of the local static area for this module and the global static area for the program.

instruction is a transputer primary instruction or the tokens **short** or **long** (see the overview of the **patch** directive for an explanation of these).

Example:

To obtain the offset between the local static area for this module and the start of the global static area:

```
patch 4 long staticfix
byte #20, #20, #20, #20
```

The above example patches the offset, in words, from the start of the global static area to the start of the local static area, for the current module, into the four byte hole.

size

Syntax:

size <expr> <instruction>

The **size** directive causes the instruction following to be encoded in exactly *n* bytes, where *n* is given by the expression in the directive. If the instruction can be encoded in less bytes then padding is added after the instruction to increase the size of the instruction to *n* bytes. The padding used is prefix zero instructions.

An error is reported if the size given is too small for the instruction.

Example:

To force a jump instruction to occupy 4 bytes:

```
size 4 j label
```

sourcefile

Syntax:

```
sourcefile <string>
```

Description:

The **VERSION** TCOFF record which appears at the start of all TCOFF object files contains a string which holds the name of the source file used to create the object file. The **sourcefile** directive causes this name to be changed to the name given by its string operand.

If no **sourcefile** directive appears in the input then the source file name defaults to the filename given on the command line by the user.

Example:

To enter the filename **fred.s**:

```
sourcefile "fred.s"
```


textname

Syntax:

textname <string>

Description:

The **textname** directive replaces the default code section name for the current module with the name given in the **string**. This is required in order to perform priority linkage (see chapter 9).

If no **textname** directive appears in the input then the text section name defaults to **text%base**.

Example:

To change the name of the text section to **fred**:

```
textname "fred"
```

toolname

Syntax:

toolname <string>

Description:

The **VERSION** TCOFF record which appears at the start of all TCOFF object files contains a string which holds the name of the tool used to create the object file. The **toolname** directive causes this name to be changed to the name given by its string operand.

If no **toolname** directive appears in the input then the tool name string defaults to 'iasm'.

Example:

To set the tool name to be **fred**:

```
toolname "fred"
```

word

Syntax:

word <expr> [, <expr>]

Description:

The **word** directive generates a sequence of words containing the values of the expressions. Each word is stored in little-endian format.

Example:

```
word 3, 3 + 4
```

This stores the values 3 and 7 to the next two consecutive words in the code section.

C.5 BNF grammar for assembler language

```

assembler-file      =   line { nl line }

line                 =   [ label-def ] [ command-list ] [ comment ]

label-def           =   symbol :

command-list        =   command { separator command }

separator           =   nl
                        |   ;

command              =   tp-instruction
                        |   directive

comment              =   -- string

tp-instruction      =   primary-op expression
                        |   secondary-op

primary-op          =   <any primary instruction (in lower case)>

secondary-op        =   <any secondary instruction (in lower case)>

directive           =   align
                        |   blkb expression { , expr-or-string }
                        |   blkw expression { , expression }
                        |   byte expr-or-string { , expr-or-string }
                        |   comment string
                        |   common symbol expression
                        |   data symbol expression
                        |   debug number , number { , number-or-string }
                        |   defsymb symbol expression
                        |   descriptor symbol string language-type
                        |   expression expression string
                        |   extern symbol
                        |   global symbol
                        |   init
                        |   language language-type
                        |   maininit
                        |   map1 string
                        |   map2 string expression
                        |   patch expression patch-instruction patch-type
                        |   size expression tp-instruction
                        |   sourcefile string
                        |   textname string
                        |   toolname string
                        |   word expression { , expression }

```

<i>patch-instruction</i>	=	<i>primary-op</i> short long
<i>patch-type</i>	=	codefix <i>symbol expression</i> datafix <i>symbol expression</i> staticfix modnumber limit extoffset <i>symbol expression</i>
<i>language-type</i>	=	unknown occam ansi_c fortran iso_pascal modula2 ada assembler occam_harness
<i>expr-or-string</i>	=	<i>expression</i> <i>string</i>
<i>number-or-string</i>	=	<i>number</i> <i>string</i>
<i>expression</i>	=	<i>number</i> <i>symbol</i> <i>monadic-expression</i> <i>dyadic-expression</i> (<i>expression</i>)
<i>dyadic-expression</i>	=	<i>expression</i> <i>dyadic-operator</i> <i>expression</i>
<i>monadic-expression</i>	=	<i>monadic-operator</i> <i>expression</i>
<i>dyadic-operator</i>	=	* / % + - & ^ < >
<i>monadic-operator</i>	=	- ~

number = *decimal-number*
 | *hex-number*

decimal-number = <any base 10 number>

hex-number = *hex-intro* <any base 16 number>

hex-intro = #
 | 0x
 | 0X

symbol = <any alphanumeric characters>
 | .
 | _ (underscore)

string = " <any sequence of printable ASCII characters> "

A full list of the *primary-op* and *secondary-op* supported is given in appendix A of the *ANSI C Toolset User Guide*.

C.6 Errors

There are three levels of errors which can occur in the assembler: *Error*, *Serious* and *Fatal*. These messages adhere to the standard format for error messages produced by the toolset. This format is documented in appendix A.

C.6.1 Fatal Errors

These are runtime errors within the assembler. They usually indicate a fault in the assembler. The assembler outputs the error message followed by a banner directing users to seek support. The assembler then exits.

The banner is as follows:

```
*****
* The assembler has detected an internal inconsistency.      *
* Please contact your supplier who may be able to help      *
* you immediately and will be able to report a suspected    *
* assembler fault to Inmos Limited.                          *
*****
```

Note: that if the error occurred before the file was opened then the **filename** and **line number** are omitted.

C.6.2 Serious Errors

These are errors from which the assembler cannot recover, e.g. not being able to open a file or out of memory. The assembler will output the error message and then exit.

Note: if the error occurred before the file was opened then the **filename** and **line number** are omitted from the message.

Arithmetic overflow in expression *operator*

The expression generator has discovered an operation which may result in overflow. The only instance of this at present is the following expression:
MOSTNEG_INT / -1. Currently *operator* can only be **/**.

Attempt to divide by 0 in expression *operator*

An attempt to divide by zero has been detected in the expression generator. *operator* can be either **/** or **%**.

Cannot open *filename* for input

The file *filename* could not be opened for reading.

Cannot open *filename* for output

The file *filename* could not be opened for writing.

Cannot open VM file *filename* for output

The assembler's internal virtual memory system has tried to open a file for output and failed.

Error writing object file

A file system error has been detected while writing the object file.

Found a map directive with no mapfile

A map directive, `map1` or `map2` has been found but no map file is open.

Out of memory

The assembler is unable to allocate any more memory.

Token too large: *string*

The token *string* is greater than 1024 characters. Only the first 32 characters of the token are given.

VM file operation failure

An operation on the virtual memory file failed (One of seek, read, write).

C.6.3 Errors

These are errors which the assembler will attempt to recover from. They generally occur while reading the input file. Usually the assembler, on detecting the error, will restart assembly from the next command after the command in which the error occurred. No object file is produced if an error occurs.

***position* operand to *directive-name* must be a numeric expression**

The directive given by *directive-name* has been given an incorrect operand. The operand should be a numeric expression. Which operand is erroneous is given by *position*.

'*symbol-name*' has already been defined with DEFSYM

The symbol *symbol-name* has been defined using `defsym` and has now been discovered in a label definition context. This is illegal.

'*symbol-name*' may not be redefined using DEFSYM

The symbol *symbol-name* has been previously defined somehow. It is now appears in a `defsym` directive. This is illegal.

Cannot mix Code and Data symbols in same expression

An expression has been discovered where code and data symbols are mixed. This is illegal.

Colon missing – assumed

The assembler was expecting a colon. It assumes one existed and continues.

Data symbol re-defined as code symbol '*symbol-name*'

The data symbol, *symbol-name*, has been redefined as a code symbol.

Debug: variable dims should be 0

The dimension field of a debug variable record should be zero. A non-zero value has been found.

Duplicate definition of symbol '*symbol-name*'

The symbol *symbol-name* has been previously defined.

Duplicate label definition '*symbol-name*'

The label given by *symbol-name* has been redefined.

Error in data mapping

A data item has been encountered twice at the mapping stage.

Expected number in *directive-name* directive got symbol

A number was expected in the directive given by *directive-name*, instead we got the symbol given by *symbol*.

Expected string in *directive-name* directive got symbol

A string was expected in the directive given by *directive-name*, instead we got the symbol given by *symbol*.

Illegal *position* operand to *directive-name* directive

The directive given by *directive-name* has been given an incorrect operand. Which operand is erroneous is given by *position*.

Illegal patch size *number*

A patch directive has been encountered with a size operand which is outside the range of legal patch sizes, less than one or greater than 255. The patch size encountered is given by *number*.

Illegal symbol in *directive-name* directive: '*symbol-name*'

The directive given by *directive-name* contains an illegal symbol (given by *symbol-name*).

instruction won't fit in *number* byte(s)

The size of an instruction requested with the **size** *directive* is too small.

Malformed expression

A badly formed expression has been encountered by the expression parser.

Only numeric expressions or strings allowed in *directive-name*

An operand to the directive, *directive-name*, (one of **byte** or **blkb**) has been found which is not a string or a number.

Symbol *symbol-name* is undefined

The symbol given by *symbol-name* has been encountered and it is undefined.

symbol in DESCRIPTOR must be global

The symbol in the *descriptor directive* must be declared as global.

Undefined symbol '*symbol-name*' as operand to *directive-name*

The symbol given by *symbol-name* is undefined and has been used in the directive given by *directive-name*. This is illegal.

Unexpected symbol: '*symbol*' *hex-number*

The symbol given by *symbol* and *hex-number* was encountered in the main loop of the assembler parser.

Unexpected symbol '*symbol*' in expression

The symbol *symbol* has been found in an expression by the expression parser. It shouldn't be there.

Value *hex-number* out of range for *directive-name directive*

The value given by *hex-number* has been encountered in the directive given by *directive-name* (one of **byte** or **blkb**) and is not in the range of a byte value.

Wrong length for *patch-type patch* – should be *number*

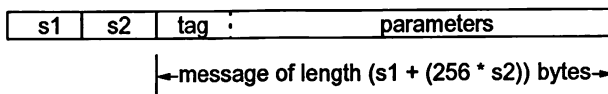
The length specified for the patch given by *patch-type* (one of **long** or **short**) is incorrect. It should be that given by *number*.

D `iserver` protocol

This appendix provides a technical description of the host file server protocol for version 1.5 of the `iserver`. It also describes the basic set of server functions which all versions of `iserver` must support and includes a set of extensions which may be present in some versions of `iserver`.

D.1 `iserver` packets

Every communication, both to and from the server, is a packet comprising a counted array of bytes. The first two bytes are a (little endian) count of the following message length. This is followed by a *tag* byte which specifies the `iserver` command. The remaining bytes are parameters to the command. Results returned by the `iserver` have a result value in place of the tag byte.



In occam this protocol is defined as:

INT16: : [] BYTE

In the to-server direction, there is a minimum packet length of 8 bytes (i.e. a minimum message length of 6 bytes). In both to and from directions there is a maximum packet length of 1040 bytes. The packet size must always be an even number of bytes. If the number of bytes is an odd a dummy byte must be added to the end of the packet and the packet byte count rounded up by one.

The server code on the host can take advantage of the fact that it will always be able to read 8 bytes from the link at the start of a transaction.

D.2 Server commands

The functions implemented by the server are separated into five groups:

- File commands
- Record Structured file commands
- Host commands
- Server commands
- Reserved and Third Party commands

The following sections contain descriptions of each command under each of the five groups.

In the descriptions the arguments and results of server calls are listed in the order that they appear in the data part of the protocol packet. The length of a packet is the length of all the items concatenated together, rounded up to an even number of bytes.

occam types are used to define the format of data items in the packet. All integer types are communicated least significant byte first. Negative integers are represented in 2s complement. Strings and other variable length blocks are introduced by a 16 bit signed count.

All server calls return a result byte as the first item in the return packet. If the operation succeeds the result byte will be zero. If the operation fails the result byte will be non-zero. The result will be one (1) in the special case where the operation failed because it was not implemented. If the result is not zero, some or all of the return values may not be present, resulting in a smaller return packet than if the call was successful. All server calls will use, where possible, a failure code from Table D.1 to give details of the failure.¹

Value	Name	Description
0	Success	Success.
1	NoCommand	Command not implemented.
128	Reserved	Unknown error.
129	Failed	Unknown error.
130	Reserved	Never generated.
131	NoPriv	Insufficient privilege.
132	NoResource	Insufficient system resources available.
133	NoFile	File not found.
134	Truncated	Data truncated.
135	BadId	A bad file id was specified.
136	NoPosn	File position has been lost.
137	NotAvailable	The requested configuration can not be provided.
138	EOF	An end of file mark has been encountered.
139	Reserved	Reserved for use by Linkops.
140	Reserved	Reserved for use by Linkops.
141	BadParas	Invalid or inconsistent parameters.

Table D.1 `iserver` failure codes

1. Result values between 2 and 127 are defined to have particular meanings by occam server libraries, result values of 128 or above are specific to the implementation of a server.

D.3 File commands

Open files are identified with 32 bit descriptors. There are three predefined open files:

```

0  standard input
1  standard output
2  standard error

```

If one of these is closed it may not be reopened.

If an application is both reading and writing to a file, then no read operation can be followed directly with a write operation and vice versa. **Fseek** must be called between a read/write or write/read, otherwise the error code **NoPosn** will be returned.

When reading from a file open in **text** mode, the host's newline sequence will be translated into the single character **LINEFEED(0x0a)**. No translation will be performed on binary files.

When writing to a file open in **text** mode, the single character **LINEFEED (0x0a)** will be translated into the host's newline sequence. No translation will be performed on binary files.

D.3.1 Fopen – Open a file

Synopsis: **StreamId = Fopen(Name, Type, Mode)**

```

To server:    BYTE                    Tag = 10
                 INT16::[]BYTE        Name
                 BYTE                    Type = 1 or 2
                 BYTE                    Mode = 1...6

```

```

From server: BYTE                    Result
                 INT32                  StreamId

```

Fopen opens the file **Name** and, if successful, returns a stream identifier **StreamId**.

Type can take one of two possible values:

Value	Name	Description
1	Binary	The file will contain raw binary bytes.
2	Text	The file will be stored as text records. Text files are host-specified.

Mode can have 6 possible values:

Value	Name	Description
1	Input	Open an existing file for input
2	Output	Create a new file, or truncate an existing one, for output
3	Append	Create a new file, or append to an existing one, for output
4	ExistingUpdate	Open an existing file for update (both reading and writing), starting at the beginning of the file
5	NewUpdate	Create a new file, or truncate an existing one, for update
6	AppendUpdate	Create a new file, or append to an existing one, for update

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions however; an output operation may not follow an input operation without an intervening **Fseek**, **Ftell** or **Fflush** operation.

The number of streams that may be open at one time is host-specified, but will not be less than eight (including the three predefines).

Return Codes

Success Failed NoPriv NoResource NoFile

D.3.2 Fclose – Close a file

Synopsis: **Fclose(StreamId)**

To server: **BYTE** **Tag = 11**
 INT32 **StreamId**

From server: **BYTE** **Result**

Fclose closes a stream **StreamId** which should be open for input or output. **Fclose** flushes any unwritten data and discards any unread buffered input before closing the stream.

Return Codes

Success Failed BadId NoResource

D.3.3 Fread – Read a block of data

Synopsis: `Data = Fread(StreamId, Count)`

To server: `BYTE Tag = 12`
 `INT32 StreamId`
 `INT16 Count`

From server: `BYTE Result`
 `INT16::[]BYTE Data`

This function should not be used. It has been included only for compatibility purposes. A replacement routine, `FGetBlock`, has been provided, which is described in D.3.5.

`Fread` reads `Count` bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If `Count` is less than one then no input is done. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

`Result` is always zero. The actual number of bytes returned may be less than requested and `Feof` and `FerrStat` should be used to check for status.

Return Codes

Success

D.3.4 Fwrite – Write a block of data

Synopsis: `Written = Fwrite(StreamId, Data)`

To server: `BYTE Tag = 13`
 `INT32 StreamId`
 `INT16::[]BYTE Data`

From server: `BYTE Result`
 `INT16 Written`

This function should not be used. It has been included only for compatibility purposes. A replacement routine, `FPutBlock`, has been provided, which is described in D.3.6.

`Fwrite` writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of `Data` is less than zero then no output is done. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position is undefined.

Fwrite returns the number of bytes actually output in **Written**. **Result** is always zero. The actual number of bytes returned may be less than requested and **Feof** and **FerrStat** should be used to check for status.

If the **StreamId** is 1 (standard output) then the write is automatically flushed.

Return Codes

Success

D.3.5 FGetBlock – Read a block of data and return success

Synopsis: **Data = FGetBlock(StreamId, Count)**

To server: **BYTE** **Tag = 23**
 INT32
 INT16 **Count**

From server: **BYTE** **Result**
 INT16::[]BYTE **Data**

FGetBlock reads **Count** bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If **Count** is less than one then no input is done. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

The actual number of bytes returned may be less than requested. This is considered a failure. **Result** will contain 0 to indicate success, anything else failure, in which case **Feof** and **FerrStat** should be used to check for status.

This function is preferred over the *Fread* function which should no longer be used.

Return Codes

Success Failed Truncated BadId

D.3.6 FPutBlock – Write a block of data and return success

Synopsis: **Written = FPutBlock(StreamId, Data)**

To server: **BYTE** **Tag = 24**
 INT32 **StreamId**
 INT16::[]**BYTE** **Data**

From server: **BYTE** **Result**
 INT16 **Written**

FPutBlock writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of **Data** is less than or equal to zero then no output is done. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position is undefined.

FPutBlock returns the number of bytes actually output in **Written**. The actual number of bytes returned may be less than requested. **Result** will contain 0 to indicate success, anything else failure, in which case **Feof** and **FerrStat** should be used to check for status.

If the **StreamId** is 1 (standard output) then the write is automatically flushed.

This function is preferred over the *Fwrite* function which should no longer be used.

Return Codes

Success **Failed** **NoResource** **BadId** **NoPosn**

D.3.7 Fgets – Read a line

Synopsis: **Data = Fgets(StreamId, Count)**

To server: **BYTE** **Tag = 14**
 INT32 **StreamId**
 INT16 **Count**

From server: **BYTE** **Result**
 INT16::[]**BYTE** **Data**

Fgets reads a line from a stream which must be open for input. Characters are read until end of file is reached, a newline is seen or the number of characters read is equal to **Count**.

If the input is terminated because a newline is seen then the newline sequence is *not* included in the returned array.

If end of file is encountered and nothing has been read from the stream then **Fgets** fails.

Return Codes

Success Failed BadId NoPosn

D.3.8 Fputs – Write a line

Synopsis: **Fputs(StreamId, String)**

To server: **BYTE** **Tag = 15**
 INT32 **StreamId**
 INT16::[]BYTE **String**

From server: **BYTE** **Result**

Fputs writes a line of text to a stream which must be open for output. The host-specified convention for newline will be appended to the line and output to the file. The maximum line length is host-specified.

Return Codes

Success Failed NoResource BadId NoPosn

D.3.9 Fflush – Flush a stream

Synopsis: **Fflush(StreamId)**

To server: **BYTE** **Tag = 16**
 INT32 **StreamId**

From server: **BYTE** **Result**

Fflush flushes the specified stream, which should be open for output. Any internally buffered data is written to the destination device. The stream remains open.

Return Codes

Success Failed NoResource BadId

D.4 Record Structured file commands

This section describes the commands for record structured files. File formats are discussed in Section D.8.

D.4.1 FopenRec – Open a record structured file

Synopsis: `StreamId, RealMrs = FopenRec(Name, Type, Mode, Mrs)`

```

To server:    BYTE          Tag = 26
               INT16::[]BYTE Name
               BYTE          Organisation = 3...4
               BYTE          Mode = 1...6
               BYTE          Type = 0...2
               BYTE          Format = 0...1
               INT32         Mrs

From server:  BYTE          Result
               INT32         RealMrs
               INT32         StreamId
               BYTE          RealType
  
```

FopenRec opens the file `Name` and, if successful, returns a stream identifier `StreamId`.

Organisation can take one of two possible values:

Value	Name	Description
3	Variable	The file will be organized in records of variable length. The maximum record size is contained in the <code>Mrs</code> field.
4	Fixed	The file will be organized in records of fixed length. The size of a record is supplied in the <code>Mrs</code> field. Record files are implemented in a host-specific way.

For each organization, **Type** can have one of the following values:

Value	Name	Description
0	DontCare	Use whatever type is most natural, and return the type in <code>RealType</code> .
1	Stream	Use streams.
2	Record	Use record oriented files.

Format can take one of two possible values:

Value	Name	Description
0	Formatted	Formatted record structured file
1	Unformatted	Unformatted record structured file

Mode can have 6 possible values:

Value	Name	Description
1	Input	Open an existing file for input
2	Output	Create a new file, or truncate an existing one, for output
3	Append	Create a new file, or append to an existing one, for output
4	ExistingUpdate	Open an existing file for update (both reading and writing), starting at the beginning of the file
5	NewUpdate	Create a new file, or truncate an existing one, for update
6	AppendUpdate	Create a new file, or append to an existing one, for update

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions however; an output operation may not follow an input operation without an intervening **Fseek**, **Ftell** or **Fflush** operation.

When an existing file is opened, the record size supplied in the open request is compared with that stored in the file if possible (some hosts do not record this information). If they are different then the open fails. If the real record size is not available, it is considered to be the same as the requested record size, and so the open can succeed. The actual record size for the file is returned in **RealMrs**.

The number of streams that may be open at one time is host-specified, but will not be less than eight (including the three predefines).

Return Codes

Success Failed NoPriv NoResource NoFile
NotAvailable

D.4.2 FGetRec – Read a record

Synopsis: **Data = FGetRec(StreamId)**

To server:	BYTE	Tag = 27
	INT32	StreamId
	INT16	ChunkSize
	INT32	Offset
	BYTE	PerformRead

From server:	BYTE	Result
	INT32	RecordSize
	INT16::[]BYTE	Data

FGetRec reads a record from a record stream which must be open for input.

If an end of file record is encountered then FGetRec fails. If PerformRead is non-zero, then a record is transferred from the file specified by StreamId into the server's buffer, otherwise, data from the previous FGetRec is transferred. If PerformRead is zero, and no records have ever been read from StreamId, then FGetRec fails. ChunkSize specifies the number of bytes to be transferred, starting at byte number Offset from the record buffer.

Return Codes

Success	Failed	BadId	NoPosn	EOF
---------	--------	-------	--------	-----

D.4.3 FPutRec – Write a record

Synopsis: **FPutRec(StreamId, Record)**

To server:	BYTE	Tag = 28
	INT32	StreamId
	INT32	RecordSize
	INT16	ChunkSize
	INT32	Offset
	BYTE	PerformWrite
	INT16::[]BYTE	Record

From server:	BYTE	Result
---------------------	-------------	---------------

FPutRec writes a record to a record stream which must be open for output. ChunkSize specifies the number of bytes to be transferred, starting at byte number Offset into the record buffer. RecordSize specifies the size of the entire record. If PerformWrite is non-zero, then a record is transferred to the file specified by StreamId from the server's buffer, otherwise, data from a previous record is transferred.

Return Codes

Success Failed BadId NoPosn

D.4.4 FputEOF – Write an end of file record

Synopsis: **FputEOF(StreamId)**

To server: **BYTE** **Tag = 29**
 INT32 **StreamId**

From server: **BYTE** **Result**

FPutEOF writes an end of file record to a record structured file. When the file is closed, the file will be truncated after the first end-of-file record and data after it will be lost.

Return Codes

Success Failed BadId

D.4.5 Fseek – Set position in a file

Synopsis: **Fseek(StreamId, Offset, Origin)**

To server: **BYTE** **Tag = 17**
 INT32 **StreamId**
 INT32 **Offset**
 INT32 **Origin**

From server: **BYTE** **Result**

Fseek sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be **Offset** characters from **Origin** which may take one of three values:

Value	Name	Description
1	Set	The beginning of the file.
2	Current	The current position in the file.
3	End	The end of the file.

For a text stream, **Offset** must be zero or a value returned by **Ftell**. If the latter is used then **Origin** must be set to 1. For a record structured file **Offset** is the number of records to seek from **Origin**.

Return Codes

Success Failed BadId

D.4.6 Ftell – Find out position in a file

Synopsis: **Position = Ftell(StreamId)**

To server: **BYTE Tag = 18**
 INT32 StreamId

From server: **BYTE Result**
 INT32 Position

Ftell returns the current file position for **StreamId**. For record structured files, the **Position** is the record number relative to the start of the file.

Return Codes

Success Failed BadId

D.4.7 Feof – Test for end of file

Synopsis: **Feof(StreamId)**

To server: **BYTE Tag = 19**
 INT32 StreamId

From server: **BYTE Result**

Feof succeeds if the end of file indicator for **StreamId** is set. Note that the definition of 'end of file' is any attempt to read past the last byte in the file. i.e. Reading the last byte in a file will not set EOF; attempting to read the next byte will.

Return Codes

Success Failed BadId

D.4.8 Ferror – Get file error status

Synopsis: **ErrorNo, Message = Ferror(StreamId)**

To server: **BYTE** **Tag = 20**
 INT32 **StreamId**

From server: **BYTE** **Result**
 INT32 **ErrorNo**
 INT16::[]BYTE **Message**

This function should not be used. It has been included only for compatibility purposes. A replacement routine, **FGetBlock**, has been provided, which is described in D.3.5.

Ferror succeeds if the error indicator for **StreamId** is set. If it is, **Ferror** returns a host-defined error number and a (possibly null) message corresponding to the last file error on the specified stream. The maximum size of **Message** will be restricted to 506 bytes for compatibility purposes. If the message is longer, then it will be truncated to fit and **Ferror** will fail.

Return Codes

Success Failed

D.4.9 Remove – Delete a file

Synopsis: **Remove(Name)**

To server: **BYTE** **Tag = 21**
 INT16::[]BYTE **Name**

From server: **BYTE** **Result**

Remove deletes the named file.

Return Codes

Success Failed NoPriv NoFile

D.4.10 Rename – Rename a file

Synopsis: `Rename(OldName, NewName)`

To server: `BYTE` `Tag = 22`
 `INT16::[]BYTE` `OldName`
 `INT16::[]BYTE` `NewName`

From server: `BYTE` `Result`

`Rename` changes the name of an existing file `OldName` to `NewName`.

Return Codes

`Success` `Failed` `NoPriv` `NoFile`

D.4.11 Isatty – Discover if a stream is connected to a terminal

Synopsis: `Isatty(StreamId)`

To server: `BYTE` `Tag = 25`
 `INT32` `StreamId`

From server: `BYTE` `Result`
 `BYTE` `Istty`

`Isatty` determines if the stream specified by `StreamId` is connected to a terminal. Any non-zero value in `Isatty` indicates that the stream is connected to a terminal.

Return Codes

`Success` `Failed` `BadId`

D.4.12 FileExists – Check to see if a file exists

Synopsis: **Exists := FileExists(Name)**

To server: **BYTE** **Tag = 80**
 INT16::[]BYTE **Name**

From server: **BYTE** **Result**
 BYTE **Exists**

FileExists checks to see if the file **Name** exists. Any non-zero value in **Exists** indicates that the file does exist.

Return Codes

Success Failed

D.4.13 FerrStat – Get file error status

Synopsis: **ErrorNo, Message = FerrStat(StreamId)**

To server: **BYTE** **Tag = 82**
 INT32 **StreamId**
 INT16 **MessLen**

From server: **BYTE** **Result**
 INT32 **ErrorNo**
 INT16::[]BYTE **Message**

FerrStat succeeds if the error indicator for **StreamId** is set. If it is, **FerrStat** returns a host-defined error number and a (possibly null) message corresponding to the last file error on the specified stream. The maximum size of **Message** is restricted to **MessLen** bytes. If the message is longer, then it will be truncated to fit and **FerrStat** will fail with status of **Truncated**.

Return Codes

Success Failed Truncated

D.5 Host commands

D.5.1 Getkey – Get a keystroke

Synopsis: **Key = GetKey()**

To server: **BYTE** **Tag = 30**

From server: **BYTE** **Result**
 BYTE **Key**

GetKey gets a single character from the keyboard. The keystroke is waited on indefinitely and will not be echoed. The effect on any buffered data in the standard input stream is host-defined. It should be noted that **GetKey** will only get one character from the keyboard stream; if a single key-press results in more than one character being generated, then **GetKey/PollKey** should be called as many times as required to read them all.

Return Codes

Success Failed

D.5.2 Pollkey – Test for a key

Synopsis: **Key = PollKey()**

To server: **BYTE** **Tag = 31**

From server: **BYTE** **Result**
 BYTE **Key**

PollKey gets a single character from the keyboard. If a keystroke is not available then **PollKey** returns immediately with a non-zero result. If a keystroke is available it will not be echoed. The effect on any buffered data in the standard input stream is host-defined. It should be noted that **PollKey** will only get one character from the keyboard stream; if a single key-press results in more than one character being generated, then **GetKey/PollKey** should be called as many times as required to read them all.

Return Codes

Success Failed

D.5.3 RequestKey – Request a single keyboard 'event'

Synopsis: **Result = RequestKey()**

To server: **BYTE** **Tag = 36**

From server: **BYTE** **Result**

This command should never be generated by an application. It may be used by the linkops server to improve performance over a network.

Once iserver has received one of these requests, it monitors the keyboard and if a key is pressed, it generates a keyboard 'event' across the link. This event will never reach the application since it will have been filtered out by the linkops server. Each RequestKey command will only solicit one keyboard event. The 'event' that iserver generates looks exactly like the reply to the GetKey command. It will only be generated while iserver is idle (waiting for another iserver request to arrive).

Return Codes

Success & Failed

D.5.4 Getenv – Get environment variable

Synopsis: **Value = Getenv(Name)**

To server: **BYTE** **Tag = 32**
 INT16::[]BYTE **Name**

From server: **BYTE** **Result**
 INT16::[]BYTE **Value**

This function should not be used. It has been included only for compatibility purposes. A replacement routine, Translate, has been provided, which is described in D.5.7.

Getenv returns a host-defined environment string for Name. If Name is undefined then Result will be non-zero. If the resultant environment string for Name is longer than 509 bytes, then it will be truncated to fit and Getenv will fail.

Return Codes

Success Failed

D.5.5 Time – Get the time of day

Synopsis: `LocalTime, UTCTime = Time()`

To server: `BYTE` `Tag = 33`

From server: `BYTE` `Result`
 `UNSIGNED INT32` `LocalTime`
 `UNSIGNED INT32` `UTCTime`

`Time` returns the local time and *Coordinated Universal Time* if it is available. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero.

Return Codes

Success Failed

D.5.6 System – Run a command

Synopsis: `Status = System(Command)`

To server: `BYTE` `Tag = 34`
 `INT16::[]BYTE` `Command`

From server: `BYTE` `Result`
 `INT32` `Status`

`System` provides access to the host's command processor, if one is available. If the length of `Command` is zero, then the command processor will not be invoked (and the empty string therefore not executed), and `System` will succeed only if a command processor is available. The value of `Status` is undefined in this case. If the length of `Command` is non-zero, then the string is passed to the command processor, which will attempt to execute it. In this case `Status` is the return value of the command, which is host-defined.

Return Codes

Success Failed NoResource

D.5.7 Translate – Translate an environment variable

Synopsis: **Value = Translate(Name, Length)**

To server:	BYTE	Tag = 81
	INT32	Offset
	INT16	Length
	INT16::[]BYTE	Name
From server:	BYTE	Result
	INT32	TotalLength
	INT16::[]BYTE	Value

Translate returns a host-defined environment string for **Name**. If **Name** is undefined then **Result** will be non-zero. Data is transferred from the resultant string starting at **Offset** for **Length** bytes. If **Offset** is beyond the end of the string, then an empty (zero length) string will be returned. The **TotalLength** field of the reply contains the total length of the translated string.

Return Codes

Success Failed

D.6 Server commands

D.6.1 Exit – Terminate the server

Synopsis: **Exit(Status)**

To server: **BYTE** **Tag = 35**
 INT32 **Status**

From server: **BYTE** **Result**

Exit terminates the server, which exits returning **Status** to its caller.

If **Status** has the special value 999999999 then the server will terminate with a host-specific 'success' result.

If **Status** has the special value -999999999 then the server will terminate with a host-specific 'failure' result.

Return Codes

Success

D.6.2 CommandLine – Retrieve the server command line

Synopsis: **String = CommandLine(All)**

To server: **BYTE** **Tag = 40**
 BYTE **All**

From server: **BYTE** **Result**
 INT16::[]BYTE **String**

This function should not be used. It has been included only for compatibility purposes. A replacement routine, **CommandArgs**, has been provided, which is described in D.6.6.

CommandLine obtains the command line passed to the server. The server is passed the command line arguments as a number of discrete items. The items are built into a command string using the following rules.

- The individual items are concatenated together into a string, with a single space (0x20) being inserted between each item.
- Any quote character (0x22) found in any item is quoted. e.g. " becomes "".
- Any command line item found to contain whitespace (0x20 or 0x09) has a quote character prefixed to it and another added after it.

CommandLine returns the command line passed to the server on invocation. On certain operating systems it is possible to quote arguments on the command line. The quotes themselves have been removed by the time **iserver** gets to see the arguments. When building the command line to pass on to the application, **iserver** places quotes (0x22) around any argument containing whitespace. Any genuine quote characters in the command line are quoted. e.g. " becomes "".

If **All** is zero the returned string is the command line, with the server name, arguments that the server recognized (including any parameters to the arguments) removed.

If **All** is non-zero then the string returned is the entire command vector as passed to the server on startup, including the name of the server command itself.

Return Codes

Success Failed

D.6.3 Core – Read peeked memory

Synopsis **Data = Core(Offset, Length)**

To server:	BYTE	Tag = 41
	INT32	Offset
	INT16	Length

From server:	BYTE	Result
	INT16: : []BYTE	Core

Core returns the contents of the root transputer's memory, as peeked from the transputer when the server was invoked with the analyze option.

Core fails if **Offset** is larger than the amount of memory peeked from the transputer or if the transputer was not analyzed.

If (**Offset + Length**) is larger than the total amount of memory that was peeked then as many bytes as are available from the given offset are returned.

If **Offset** and **Length** are both zero, the the result of this function will indicate if the transputer was analyzed and peeked by the server.

Return Codes

Success Failed

D.6.4 Version – Find out about the server

Synopsis: **Id = Version()**

To server: **BYTE** **Tag = 42**

From server: **BYTE** **Result**
 BYTE **Version**
 BYTE **Host**
 BYTE **OS**
 BYTE **Board**

Use of this function is discouraged. To obtain similar information in a more portable manner, use the function **GetInfo** (D.6.5).

Version returns four bytes containing identification information about the server and the host it is running on.

If any of the bytes has the value 0 then that information is not available.

Version identifies the server version. The byte value should be divided by ten to yield a version number

Host identifies the host box. Currently 8 are defined:

Value	Host	Value	Host
1	PC	2	NEC-PC
3	VAX	4	Sun 3
5	IBM 370	6	Sun 4
7	Sun 386i	8	Apollo

OS identifies the host environment. Currently 5 are defined:

Value	Operating system	Value	Operating system
1	DOS	2	Helios
3	VMS	4	SunOS
5	CMS		

Board identifies the interface board. Currently 12 are defined:

Value	Board	Value	Board
1	IMS B004	2	IMS B008
3	IMS B010	4	IMS B011
5	IMS B014	6	DRX-11
7	Caplin QT0	8	IMS B015
9	IBM CAT	10	IMS B016
11	UDP	12	TCPlink

INMOS reserves numbers up to and including 127 for these three fields.

Return Codes

Success Failed

D.6.5 GetInfo – Obtain information about the host and server

Synopsis: NoOfBytes = GetInfo(Item, Buffer)

To server: BYTE Tag = 43
 BYTE Item
 INT16 ReplySize

From server: BYTE Result
 Item specific result

GetInfo is used to obtain host and server specific information in as portable a fashion as possible. **ReplySize** specifies the maximum size of the reply in bytes. If the reply exceeds this value, it will be truncated and an appropriate failure status will be returned. Values for **Item** and their representation are shown in Table D.2.

Value	Name	Returned as	Description
1	SwitchChar	BYTE	The switch character used by the host.
2	EndOfLine	INT16: : []BYTE	The end of line sequence.
3	Stderr	BOOL	A boolean value indicating whether or not the standard error stream can be redirected.
4	ServerId	INT16: : []BYTE	A string identifying the server.
5	ServerMaj	INT32	The server's major version number.
6	ServerMin	INT32	The server's minor version number.
7	PacketSize	INT32	The server's maximum packet size.

Table D.2 Results of GetInfo command

A **BOOL** will be represented as a single byte, with any non-zero value meaning **TRUE**, and zero **FALSE**.

Return Codes

Success Failure NoPriv Truncated

D.6.6 CommandArgs – Retrieve the server command line arguments

Synopsis: **String** = **CommandArgs**(**Argno**, **Length**)

To server: **BYTE** **Tag** = 83
 INT16 **Argno**
 INT16 **Length**

From server: **BYTE** **Result**
 INT16 **NumArgs**
 BYTE **ServerArg**
 INT16::[]**BYTE** **String**

CommandArgs provides access to the server's command line. **Argno** specifies the command line argument number and is used as an index into the server's **argv** array. **Length** specifies the maximum length of **String**. If **String** is too big, it will be truncated to fit, and a status of **Truncated** will be returned. If **ServerArg** is non-zero, it indicates that **String** was recognized as a server argument, and should be ignored by the application. **NumArgs** is the highest argument number which may be requested, and is the equivalent of the C variable **argc**. Argument number zero is always present.

Return Codes

Success Failed Truncated

D.7 Reserved Tags and Third Party Tags

The following tags have been reserved for specific applications or by Third Parties for use in their own servers. Their use is not encouraged, since the third party tags will not be present in the standard INMOS server.

D.7.1 MSDOS – Perform MS-DOS specific function

Synopsis: MsDos (Command)

To server: BYTE Tag = 50
 BYTE Function code
 Function specific data

From server: BYTE Result
 Function specific results.

MSDOS is used to perform a number of MS-DOS specific functions. This is used to support some early INMOS PC-based programs. Use of this function is discouraged for portability reasons. The functions supported are shown in Table D.3.

Value	Name	Description
0	SendBlock	Write a block of data anywhere in the PC's memory map.
1	GetBlock	Read a block of data from anywhere in the PC's memory map.
2	CallInt	Invoke a software interrupt.
3	GetRegs	Read the segment registers.
4	PortWrite	Write to a port.
5	PortRead	Read from a port.

Table D.3 MS-DOS functions

Return Codes

Success Failed

D.7.2 SocketA – make a socket library call

Synopsis: **BYTE** **Tag = 70**
 BYTE **Socket operation**
 Function specific data.

From server: **BYTE** **Result**
 Function specific results.

This function allows an application to make a socket library call.

Return Codes

Success Failed

D.7.3 SocketM – make a socket library call

Synopsis: **BYTE** **Tag = 71**
 BYTE **Socket operation**
 Function specific data.

From server: **BYTE** **Result**
 Function specific results.

This function allows an application to make a socket library call.

Return Codes

Success Failed

D.7.4 ALSYS – Perform Alsys specific function

Synopsis: **AlSys (...)**

To server: **BYTE** **Tag = 100**
 Function specific data

From server: **BYTE** **Result**
 Function specific results

Alsys is used to perform a number of Alsys specific functions. This is used to support the Alsys compilers.

D.7.5 KPAR – Perform Kpar specific function

Synopsis: Kpar(...)

To server: BYTE Tag = 101
Function specific data

From server: BYTE Result
Function specific results

This is used to perform a number of Kpar specific functions. This is used to support Kpar tools.

D.8 Record Structured file format

Under VAX/VMS, record structured files are implemented using the VAX Record Management Service (VAX/RMS). Under all other hosts record structured files are implemented using binary files. Files created are of a similar format to that used by Sun FORTRAN.

D.8.1 SunOS and MS-DOS

Formatted Sequential

Each record in a Formatted Sequential file has a linefeed character (0x0a) appended to it. Thus an extra byte per record is required. i.e.

record data	linefeed
-------------	----------

Unformatted Sequential

Unformatted Sequential files are implemented by prefixing and suffixing each record with a four byte length field, most significant byte first. The length field is the length of the data, not the data plus the length fields. This means an extra eight bytes per record are required. i.e.

data length	length bytes of data	data length
-------------	----------------------	-------------

Formatted Direct

The record size is specified at open.

Unformatted Direct

The record size is specified at open.

D.9 Termination codes

There are various circumstances under which **iserver** can terminate. **iserver** 1.5 makes it possible for a controlling script to distinguish between the following cases

- Terminated properly on receipt of an **sps.success** token.
- Terminated properly on receipt of an **sps.failure** token.
- Terminated properly with any other value.
- Terminated on receipt of a user break.
- Terminated on seeing the transputer error flag set.
- Any other termination.

The values used are shown in Table D.4.

Host	Termination			User Break	Error Flag	Any Other
	Success	Failure	Other			
MS-DOS	0	255	exit code	254	253	252
Helios	0	255	exit code	254	253	252
VAX/VMS	1	4	exitcode × 16	10	2	12
error class	success	fatal	warning	information	error	fatal
SunOS	0	255	exit code	254	253	252

Table D.4 **iserver** termination codes

The values chosen for VAX/VMS have been designed to generate different 'classes' of error.

Under all operating systems apart from VAX/VMS, error codes between 240 and 255 are reserved for use by **iserver**.

E ITERM files

This appendix describes the format of ITERM files; it is included for people who need to write their own ITERM because they are using terminals that are not supported by the standard ITERM file supplied with the toolset.

Standard ITERM files for this release are provided in the `iterms` directory, which is a subdirectory of the main toolset installation directory. These files may be used as templates and tailored to suit your own needs. It is recommended that the installation files are not changed in any way, and that modifications are only made to copies of the files.

E.1 Introduction

ITERMs are ASCII text files that describe the control sequences required to drive terminals. Screen oriented applications that use ITERM files are terminal independent.

ITERM files are similar in function to the UNIX *termcap* database and describe input from, as well as output to, the terminal. They allow applications that use function keys to be terminal independent and configurable.

Within the toolset, the ITERM file is only used by the debugger tool `idebug` and the T425 simulator tool `isim`.

A default ITERM file may be defined in the `ITERM` environment variable. For details see section E.8 and the Delivery Manual for the release.

E.2 The structure of an ITERM file

An ITERM file consists of three sections. These are the *host*, *screen* and *keyboard* sections. Sections are introduced by a line beginning with the section letters 'H', 'S' or 'K'. Case is unimportant and the rest of the line is ignored. Sections consist of a number of lines beginning with a digit. A section is terminated by a line beginning with the letter 'E'. The *host* section must appear first; other sections may appear in any order in the file. Sections must be separated by at least one blank line.

The syntax of the lines that make up the body of a section is best described in an example:

```
3:34,56,23,7.    comments
```

Each line starts with the index number followed by a colon and a list of numbers separated by commas. Each line is terminated by a full stop ('.') and anything fol-

lowing it is treated as a comment. Spaces are not allowed in the data string and an entry cannot be split across more than one line.

Comment lines, beginning with the character '#', may be placed anywhere in an ITerm file. Extra blank lines in the file are ignored.

The index numbers in each section correspond to an agreed meaning for the data. In the following sections the meaning of the data in each of the three sections is described in detail.

E.3 The host definitions

E.3.1 ITerm version

This item identifies an ITerm file by version. It provides some protection against incompatible future upgrades.

e.g. 1:2.

E.3.2 Screen size

This item allows applications to find out the size of the terminal at startup time. The data items are the number of columns and rows, in that order, available on the current terminal.

e.g. 2:80,25.

Screen locations should be numbered from 0, 0 by the application. Terminals which use addressing from 1, 1 can be compensated for in the definition of goto X, Y.

E.4 The screen definitions

The lists of values in the screen section represent control codes that perform certain operations; the data values are ASCII codes to send to the display device.

ITerm version 2 defines the indices given in table E.1. These definitions are used in the example ITerm file; for a complete listing of the file see section E.8.

Index	Screen operation	Index	Screen operation
1	cursor up	9	clear to end of screen
2	cursor down	10	insert line
3	cursor left	11	delete line
4	cursor right	12	ring bell
5	goto x y	13	home and clear screen
6	insert character	20	enhance on (not used)
7	delete character at cursor	21	enhance off (not used)
8	clear to end of line		

Table E.1 ITERM screen operations

For example, an entry like: '8:27,91,75.' indicates that an application should output the ASCII sequence 'ESC [K' to the terminal output stream to clear to end of line.

E.4.1 Goto X Y processing

The entry for 5, 'goto X Y', requires further interpretation by the application. A typical entry for 'goto X Y' might be:

5:27,-11,32,-21,32

The negative numbers relate to the arguments required for X and Y.

..., -ab, nn, ...

where: a is the argument number (i.e. 1 for X, 2 for Y).

b controls the data output format.

If b=1 output is an ASCII byte (e.g. 33 is output as !).

If b=2 output is an ASCII number (e.g. 33 is output as 33).

nn is added to the argument before output.

As a complete example, consider the following ITERM entry in the screen section:

5:27,91,-22,1,59,-12,1,72. ansi cursor control

This would instruct an application wishing to move the terminal cursor to X=14, Y=8 (relative to 0,0) to output the following bytes to the screen:

```
Bytes in decimal: 27  91  57  59  49  53  72
Bytes in ASCII:  ESC [  9  ;  1  5  H
```

E.5 The keyboard definitions

Each index represents a single keyboard operation. The data specified after each index defines the keystroke associated with that operation. Multiple entries for the same index indicate alternative keystrokes for the operation.

ITERM version 2 defines the indices given in table E.2. These definitions are used in the example ITERM file; for a complete listing of the file see section E.8.

Index	Function	Index	Function
2	delete character	39	goto line
6	cursor up	40	backtrace
7	cursor down	41	inspect
8	cursor left	42	channel
9	cursor right	43	top
12	delete line	44	retrace
14	start of line	45	relocate
15	end of line	46	info
18	line up	47	modify
19	line down	48	resume
20	page up	49	monitor
21	page down	50	word left
26	enter file	51	word right
27	exit file	55	top of file
28	refresh	56	end of file
29	change file	62	toggle hex
31	finish	65	continue from
34	help	66	toggle breakpoint
36	get address	67	search

Table E.2 ITERM key operations

E.6 Setting up the ITERM environment variable

To use an ITERM the application has to find and read the file. An environment variable (or logical name on VMS) called **ITERM** should be set up with the pathname of the file as its value. For example, under MS-DOS the command would be:

```
C:\> set ITERM=C:\ITools\Tools\PCANSI.ITM
```

Under Unix you would set an environment variable. For example, the command for `csb` users might be:

```
% setenv ITERM ~/.item
```

Under VMS you would define a logical name. For example:

```
$ DEFINE ITERM SYS$LOGIN:VT100.ITY
```

For more details about setting environment variables see the Delivery Manual that accompanies the release.

E.7 Iterms supplied with a toolset

The following ITERM files are supplied with the toolset:

File	Description
<code>ansi.itm</code>	Generic ANSI item
<code>ncd.itm</code>	NCD X terminal item
<code>necansi.itm</code>	NEC PC item
<code>pcansi.itm</code>	PC item (requires ANSI.SYS)
<code>sun.itm</code>	SunView item
<code>vt100.itm</code>	vt100 item

Table E.3 ITERM files supplied

`ansi.itm` is likely to be the most portable in that it will work unchanged with most hosts. However, because of this it may only use the normal (alpha-numeric keys) of a keyboard. This means that some keys (when used in conjunction with the **CNTL** or **SHIFT** key) are associated with more than one operation. Specific host items make use of known function keys etc. which leads to less overloading of keys.

Each item file may be treated as an example; you may create and use your own item file if you wish.

E.8 An example ITEM

This is the generic toolset ITEM file for an ANSI terminal.

```
# -----
#
# ANSI ITEM for any ANSI terminal
# Support for idebug and isim
#
# V1.0 16 November 1990      (RD) Created
# V1.1 11 January 1991      (NH) Modified
# -----

host section
1:2.                                version
2:80,24.                            screen size
end of host section

# screen control characters

screen section
#
# DEBUGGER      SIMULATOR
1:27,91,65.     cursor up
2:27,91,66.     cursor down
3:27,91,68.     cursor left
4:27,91,67.     cursor right
5:27,91,-22,1,59,-12,1,72. goto x y
#6.             insert char
#7.             delete char
8:27,91,75.     clear to eol
9:27,91,74.     clear to eos
#10             ansi terminals do insert line
#11             not have these delete line
12:7.           bell
13:27,91,50,74. clear screen
end of screen section

keyboard section
#
# KEY           DEBUGGER      SIMULATOR
2:127.         # DELETE      del char
2:8.           # BACKSPACE   del char
6:27,91,65.    # UP          cursor up      cursor up
7:27,91,66.    # DOWN        cursor down    cursor down
8:27,91,68.    # LEFT        cursor left     cursor left
9:27,91,67.    # RIGHT       cursor right    cursor right
12:21.         # Ctrl-U      delete line
14:1.         # CTRL-A      start of line  start of line
15:5.         # CTRL-E      end of line   end of line
18:27,85.     # ESC U        line up
18:27,117.    # ESC u        line up
19:27,68.     # ESC D        line down
19:27,100.    # ESC d        line down
20:27,86.     # ESC V        page up        page up
20:27,118.    # ESC v        page up        page up
21:27,87.     # ESC W        page down      page down
21:27,119.    # ESC w        page down      page down
26:14.        # CTRL-N      enter file
```

27:24.	# CTRL-X	exit file	
28:12.	# CTRL-L	refresh	refresh
28:23.	# CTRL-W	refresh	refresh
29:27,70.	# ESC F	change file	
29:27,102.	# ESC f	change file	
31:27,88.	# ESC X	finish	
34:27,72.	# ESC H	help	help
34:27,104.	# ESC h	help	help
36:27,65.	# ESC A	get address	
36:27,97.	# ESC a	get address	
39:7.	# CTRL-G	goto line	
40:27,48.	# ESC O	backtrace	
41:27,49.	# ESC I	inspect	
41:9.	# CTRL-I	inspect	
42:27,50.	# ESC 2	channel	
43:27,51.	# ESC 3	top	
44:27,52.	# ESC 4	retrace	
45:27,53.	# ESC 5	relocate	
46:27,54.	# ESC 6	info	
47:27,55.	# ESC 7	modify	
48:27,56.	# ESC 8	resume	
49:27,57.	# ESC 9	monitor	
50:11.	# CTRL-K	word left	
51:16.	# CTRL-P	word right	
55:27,60.	# ESC <	top of file	
56:27,62.	# ESC >	end of file	
62:27,116.	# ESC t	toggle hex	
62:27,84.	# ESC T	toggle hex	
65:27,67.	# ESC C	continue from	
65:27,99.	# ESC c	continue from	
66:2.	# CTRL-B	toggle break	
67:6.	# CTRL-F	search (Find)	

end of keyboard section

idebug key that isn't really part of item but its here all the
same!

INTERRUPT CTRL A -- IDEBUG

THAT'S ALL FOLKS

F Bootstrap loaders

F.1 Introduction

Special loading procedures can be created for the program and used in place of, or in addition to, the standard INMOS bootstrap. The file containing the new bootstrap is specified by invoking the collector with the 'B' and 'T' options.

User defined bootstraps must perform all the necessary operations to initialize the transputer, load the network, and set up the software environment for the application program.

Bootstraps are output to the program bootable file as the first section of code in the bootable file. The bootstrap, consisting of the primary and secondary bootstrap sequences, is followed by the standard INMOS network loader program, which is output in small packets, each packet consisting of a maximum of 60 bytes. The last packet of the network loader is followed by a length byte of zero.

In most cases a custom bootstrap will interface directly with the standard INMOS Network Loader, which places various pieces of code and data within the transputer memory in a controlled way. However, it is possible to skip the standard loader by sinking its code packets and following the commands used by the network loader that are output after the network loader.

The general format of a custom bootstrap is a concatenated sequence of bootstrap code segments each preceded by a length byte. The sequence can be any length. The bootstrap program must be contained in a single file.

The source of the standard INMOS Network Loader is supplied with the toolset and is fully commented. See the accompanying Delivery Manual for details of source directories supplied.

F.1.1 The example bootstrap

The example bootstrap loader provided on the toolset `examples/userboot` directory is a combination of several files used in the standard INMOS bootstrap scheme. The files have been combined into a single file to illustrate how to create a user-defined bootstrap; the functionality is the same as that used in the standard INMOS scheme based on multiple files.

The program is written in transputer code and consists of two parts:

- *Primary* bootstrap – performs processor setup operations such as initializing the transputer links

- *Secondary bootstrap* – sets up the software environment and interfaces to the Network Loader.

Transfer of control

The calling sequence in the standard INMOS scheme is as follows:

The primary loader calls the secondary loader, which then calls the Network Loader. When the Network Loader has completed its work control returns to the secondary loader, which calls the application program via data set up by the Network Loader.

Custom bootstraps should follow the same sequence.

F.1.2 Writing bootstrap loaders

Bootstrap loader programs should be written to perform the same operations as the standard scheme, that is, hardware initialization, setting up the software environment, and calling the Network Loader. If you skip the Network Loader by sinking its code bytes then you must ensure its function is reproduced in your own code. If you do use the Network Loader you must ensure the interface to it is correct by setting up the invocation stack. The method by which this is achieved can be deduced from the example program source.

If you wish to make only a few small changes to the standard loader, for example, insert code to initialize some D-to-A convertors, then the example code can be used and the required code can be inserted between the Primary and Secondary Loader code as an additional piece of bootstrap code in the sequence of bootstraps. The rest of the code can be used as it stands.

If you decide to devise your own loading scheme and rewrite the Primary and Secondary Loaders then you should be familiar with the design of the Transputer and its instruction set. For engineering data about the transputer consult the '*Transputer Databook*' and for information about how to use the instruction set see the '*Transputer Instruction Set: a compiler writer's guide*'.

Index

Symbols

- !, idebug, 135, 138, 145, 159
 - ::, idebug, 149
 - #
 - idebug, 119
 - idump, 175
 - isim, 306
 - #alias, 220
 - #define
 - linker directive, 221
 - syntax, 12
 - #elif, syntax, 12
 - #else, 13
 - syntax, 12
 - #endif, 13
 - syntax, 12
 - #error, syntax, 13
 - #if, syntax, 13
 - #ifdef, syntax, 13
 - #ifndef, syntax, 13
 - #include
 - filename syntax, 14
 - icc directive, 14
 - linker directive, 221
 - nesting icc directives, 14
 - #line, syntax, 14
 - #mainentry, 221
 - #PRAGMA, LINKAGE, 222
 - #pragma
 - IMS_codepatchsize, 15
 - IMS_descriptor, 15, 17
 - parameters, 18
 - IMS_linkage, 15, 222
 - IMS_modpatchsize, 15
 - IMS_nolink, 15, 17
 - IMS_off, 15
 - parameters, 16
 - IMS_on, 15
 - parameters, 16
 - IMS_translate, 15
 - syntax, 14
 - #reference, 221
 - #section, 222
 - #undef, syntax, 19
 - \$
 - idebug, 119
 - idump, 175
 - %
 - idebug, 119, 150
 - imap, 273
 - isim, 306
 - @, iserver, 293
 - +, idebug, 160
 - ++, idebug, 159
 - *, idebug, 126, 150, 155, 157, 162
 - **, idebug, 157, 162
 - \, in filenames, 14
 - __asm, 21
 - __lsb, 20
 - __params, 20
- ## A
- Action strings, in makefiles, 267
 - align, 346, 347
 - Analyse, 116, 117
 - ANSI C
 - compiler, 3
 - trigraphs, 24
 - Arithmetic right shift, 8
 - Arrays, subranges, 149, 159

Assembler, 341
 directives, 346
 errors, 379
 invoking, 7, 341
 language, 343
 syntax, 376
 transputer instructions, 345

B

B004, 317
 B008, 317
 Backslash, in filenames, 14
BACKTRACE, 146
 binary. *See* **output.format**
 Binary lister, 237
 command line, 238
 errors, 251
 Binary output, **ieprom**, 202
blkb, 346, 348
blkw, 346, 349
 Block mode, **ieprom**, 203
 Block move, 21
 Boards, wiring, 108
boards.inc, 53
 Boot from link, 177
 collector memory map, 94, 97
 default collector output, 86
 Boot from ROM, 92, 177, 195
 configurer options, 52
 Bootable code, 81
bootable.file, 198
 Bootstrap
 alternatives, 93
 example, 421
 loaders, 93, 422
 Break key, 320
 Breakpoint debugging, methods,
 109

Breakpoints, 124, 308
 commands, 124
 menu, 124
 Building libraries, 211
 Built-in functions, 21
byte, 346, 350
byte.select, 200

C

C, implementation, compatibility
 issues, 8
call_without_gsb, 17
 Capability, 287, 291
 specific host, 293
CHANGE FILE, 147
 Change processor,
 debugging, 137
CHANNEL, 144
char, signedness, 8
 Character, signedness, 8
 Checking a network, 125
 Clearing error flags, 166, 321
 Code
 listing, 242
 position in memory, 54, 86, 88
 Collector
 command line, 82
 error messages, 100
 input files, 85
 output files, 85
 non-bootable, 91
 Command line, 325
 Command line options
 icc, 4, 5
 icconf, 51, 52
 icollect, 84
 idebug, 111
 iemit, 178
 ieprom, 197
 ilibr, 208
 ilink, 219

- ilist**, 239
 - imakef**, 258
 - imap**, 273
 - iserver**, 285
 - isim**, 304
 - iskip**, 318
 - optimizing compiler, 6
 - specify transputer target, 339
 - comment**, 346, 351
 - Comments**
 - in assembly code, 345
 - in EPROM control files, 197
 - common**, 346, 352
 - Compare memory, debugging**, 125
 - Compatibility, other C implementa-**
tions, 8
 - Compiler**, 3
 - command line, 3
 - default, 7
 - diagnostics, 22
 - implementation data, 331
 - recoverable errors, 31
 - serious errors, 38
 - terminology, 22
 - warnings, 24
 - error modes, 7
 - memory map, 9
 - options, 4, 5, 6
 - pragmas, 15
 - predefines, 19
 - macros, 19
 - preprocessor directives, 12
 - selective loading of libraries, 210
 - Compiling, for a range of transput-**
ers, 334
 - Configuration**
 - description, example files, 53
 - language, implementation, 50
 - Configurer**, 49
 - advanced toolset options, 52
 - command line, 50
 - default command line, 52
 - diagnostics
 - recoverable errors, 60
 - serious errors, 75
 - warnings, 57
 - errors, 55
 - information messages, 56
 - memory map, 54
 - search paths, 54
 - standard definitions, 53
 - Connection database**, 292
 - example, 295
 - format, 294
 - CONTINUE FROM**, 145
 - Conventions**
 - command line options, 325
 - command line syntax, 325
 - error messages, 331
 - filenames, 326
 - imakef** file extensions, 330
 - search paths, 326
 - standard file extensions, 327
 - Core dump**, 311
 - listing, 250
 - Current location**,
 - in debugger, 146
 - Cursor positioning**, 415
- ## D
- Data, listing all**, 248
 - data**, 346, 353
 - Debug**,
 - support functions, 131, 145
 - debug**, 346, 354
 - Debugger**, 107
 - command line, 109
 - environment variables, 112
 - errors, 166
 - monitor commands
 - definitions, 123–142
 - editing functions, 120
 - mapped by ITERM, 120
 - summary, 120–122
 - monitor page
 - commands, 119
 - scroll keys, 122

- symbolic commands, 122
- program hangs, 166
- scroll keys, 119
- symbolic functions, 142
- Debugging
 - See also Monitor page
 - B004 boards, 116
 - current location, 146
 - inspecting channels, 144
 - inspecting memory, 159
 - interactive, 228
 - options,
 - for different boards, 118
 - program termination, 113
 - single step, 312
 - TRAMs, 116
- defsym, 346, 355
- DELETE, 260
- descriptor, 346, 356
- Directives
 - assembler, 346
 - linker, 220
 - preprocessor, 12
- Directory path, 326
- Disassemble memory, 126
- Display memory in hex, 129
- Display reference, 248
- Displaying object code, 237
- DRAM timing parameters, 187
- Dynamic code loading, listing files, 250

E

- Early write, 185
- Editing functions, 120
- Editing makefiles, 267
- EMI, 177
 - clock period, 185
- ENTER FILE, 123, 147

- end.offset, 200
- ENTER FILE, 147
- Environment variables, 416
 - accessing through iserver, 402
- IBOARDSize, 87
- ICCARG, 7
- ICCONFARG, 52
- ICOLLECTARG, 85
- ICONDB, 286, 293
- ILIBRARG, 208
- ILINKARG, 219
- ILISTARG, 240
- ISESSION, 286
- ISIMBATCH, 313
- ITEM, 116, 305
- TRANSPUTER, 286, 292
 - used by idebug, 112
- EPROM, 52, 92
 - code layout, 200
 - devices, 204
- EPROM program convertor, 195
 - binary output, 202
 - block mode, 203
 - command line, 196
 - control file, 197
 - errors, 206
 - hex dump, 202
 - Intel extended hex format, 203
 - Intel hex format, 203
 - Motorola S-record format, 203
 - output files, 202
- EPROM programming, 195
- eprom.space, 198
- Error
 - handling, 331
 - modes, 7, 223
 - selective loading of libraries, 210
 - runtime, 332
 - severities, 331
- Error flag
 - clearing in a network, 166, 321
 - detection in interactive debugging, 117
- Error messages
 - assembler, 379

format, 331
icc, 22
icconf, 55
icollect, 100
idebug, 166
idump, 176
iemit, 191
ieprom, 206
ilibr, 214
ilink, 230
ilist, 251
imakef, 268
imap, 281
iserver, 298
 additional, 300
isim, 314
iskip, 321

Ethernet, 283

Event, 133, 311

Examples

 bootstrap loader, 421
 configuration files, 53
 connection database, 295
 ieprom control file, 205
 imakef, 260
 occam, 263
 skipping a single processor, 319
 skipping multiple transputers, 319

EXIT FILE, 147

Exported names, listing, 244

Extensions, file, 254, 327

extern, 346, 357

External memory interface, 177

External references, listing, 251

extintel. See **output.format**

Extraction of library modules, 224

F

File

 extensions, 327
 imakef, 254, 330

imap source files, 273
 identification, 249, 327

Filename conventions, 326

FINISH, 148

G

GET ADDRESS, 147

global, 346, 358

Go to process, 129

GOTO LINE, 146

H

Heap area, 87

 position in memory, 54, 86

HELP, 122, 142, 147

hex. See **output.format**

Hexadecimal

 arguments to **idump**, 175
 listing, 244

Hexadecimal format,
 for EPROM, 202

Host

 for capability, 293
 versions, xix

Host file server, 283
 terminating, 320

I

IBOARDSIZE, 87, 113
 errors, 89

icc, 3

channel_pointers, 16
 checking
 printf, 16
 scanf, 16
 stack, 16

 command line options, 4, 5, 6
 file extension defaults, 7

- `inline_ops`, 16
- memory map, 9
- search path, 7
- syntax, 3
- ICCARG, 7
- icconf, 49
 - command line, 50
 - error messages, 55
- ICCONFARG, 52
- iccollect
 - command line, options, 84
 - command line, 82
 - environment variables, 85, 87
 - errors, 100
- ICCOLLECTARG, 85
- ICONDB, 286, 293
- idebug, 107
 - command line, 109
 - options, 111
 - environment variables, 112
 - errors, 166
 - interactive mode, 115
 - post-mortem debugging, 113
 - restarting, 115
- IDEBUGSIZE, 113
 - errors, 166
- idump, 108, 175, 287, 317
 - errors, 176
- iemit, 177
 - command line, 178
 - DRAM timing parameters, 187
 - errors, 191
 - index page, 180
 - input parameters, 182
 - memory read cycle, 188
 - memory write cycle, 189
 - timing information, 186
- ieprom, 195
 - command line, 196
 - control file, 197
 - errors, 206
- ilibr, 207, 209
 - command line, 208
- command line options, 208
- error messages, 214
- ILIBRARG, 208
- ilink, 217
 - command line, 218
 - indirect files, 219
- ILINKARG, 219
- ilist, 237
 - command line, 238
 - command line options, 239
 - errors, 251
- ILISTARG, 240
- imakef, 230, 253
 - command line, 257
 - command line options, 258
 - deleting intermediate files, 260
 - errors, 268
 - examples, 260
 - file extensions, 254, 330
 - file formats, 266
 - linker indirect files, 257, 259
 - occam examples, 263
 - target files, 254
- imap, 271
 - command line, 272
 - command line options, 273
 - errors, 281
 - output file structure, 275
- Implementation,
 - compiler diagnostics, 331
- IMS B004, 317
- IMS B008, 317
- IMS B404, 117
- IMS_descriptor, 17
- IMS_nolink, 17
- `INFO`, 144
- init, 346, 359
- Inline functions, 21
- INMOS C, implementation,
 - compatibility issues, 8
- `INSPECT`, 143

- Inspect memory, 130
- intel. *See* output . format
- Intel extended hex format, *ieprom*, 202
- Intel hex format, *ieprom*, 202
- Interactive debugging, collector option, 99
- INTERRUPT**, 145
- ISEARCH**, 14, 54, 326
- iserver**, 283, 317
 - accessing transputers, 292
 - capability, 287
 - command line, 284
 - command line options, 284
 - connection manager, 297
 - environment variables, 286
 - error codes, 298
 - error messages, 298
 - exit codes, 298
 - functions, 283
 - halt system error mode, 287
 - loading programs, 286
 - new features, 297
 - passing parameters to a program, 287
 - protocol, 383
 - file commands, 385
 - Fclose – close a file, 386
 - Feof – test for end of file, 395
 - Error – get file error status, 396
 - FerrStat – Get file error status, 398
 - Fflush – flush a stream, 390
 - FGetBlock, 388
 - FGetRec – read a record, 393
 - Fgets – read a line, 389
 - FileExists, 398
 - Fopen – open a file, 385
 - FopenRec, 391
 - FPutBlock, 389
 - FPutEOF, 394
 - FPutRec – write a record, 393
 - Fputs – write a line, 390
 - Fread – read block of data, 387
 - Fseek – set position in a file, 394
 - Ftell – find position in a file, 395
 - Fwrite – write block of data, 387
 - Isatty, 397
 - Remove – delete a file, 396
 - Rename – Rename a file, 397
- host commands
 - Getenv – get environment variable, 400
 - Getkey, 399
 - Pollkey, 399
 - System – run a command, 401
 - Time – get the time of day, 401
 - Translate – translate an environment variable, 402
- packets, 383
- record structured file commands, 391
- record structured file format, 410
- reserved commands
 - ALSYS, 409
 - KPAR, 410
 - MSDOS, 408
 - SocketA, 409
 - SocketM, 409
- server commands, 383
 - CommandArgs, 407
 - CommandLine, 403
 - Core – read peeked memory, 404
 - Exit – exit the server, 403
 - GetInfo, 406
 - Version – find out about the server, 405
- termination codes, 411
- record structured files, 298
- session manager, 284, 288, 297
 - customising interface, 290
- specifying the transputer to use, 287
- stream identifier validation, 298
- subsystem reset, 286
- terminating, 287
 - on error, 287

user interrupt, 297
ISESSION, 286, 288
isim, 303
 command line, 303
 command line options, 304
 errors, 314
ISIMBATCH, 313
iskip, 108, 317
 command line, 318
 command line options, 318
 errors, 321
ispy, 166, 321
ITERM, 113, 116, 305, 416
ITERM file
 example listing, 418
 format, 413
 keyboard, 416
 screen, 414
 use by simulator, 305, 306
 version, 414

J

JEDEC, symbol, 186, 188
 Jump instructions, in ROM, 201
 Jump into program, 131

K

Keyboard definitions, 416

L

language, 346, 360
 Late write, 185
 LFF files, listing, 250
 Librarian, 207
 command line, 208
 concatenated input, 207
 linked object input, 209
 options, 208

Library
 building, 211
 building optimized, 211
 extraction of modules, 224
 index, 207, 210
 indirect files, 207, 209
 imakef, 257
 linking supplied libraries, 220
 listing index, 246
 modules, 209
 selective loading of, 210
 usage files, 210
 imakef, 257

LINE DOWN, 123

LINE UP, 122

Link map, 228

Linker, 217
 command line, 218
 compatible transputer classes,
 222
 directives, 220
 errors, 230
 extraction of library modules, 224
 indirect files, 219
 imakef, 257, 259
 LFF output, 223
 selective loading of libraries, 210
 TCOFF output, 223

Linking, transputer targets, 333

Links, 133, 311

List. See **ilist**

Loading programs
 iserver, 283
 iskip, 320

LoadStart, 54, 55, 94, 96

localhost, 293

Location, in debugger, 146

Logical name, 416

M

Macros
 definition, 12

- in makefiles, 266
 - Main entry point, 227
 - maininit, 346, 361
 - Make programs, 253
 - Borland, 253
 - Gnu, 253
 - Microsoft, 253
 - Unix, 253
 - Makefile generator, 253
 - command line, 257
 - errors, 268
 - Makefiles
 - delete rule, 267
 - editing, 267
 - formats, 266
 - macros, 266
 - map1, 346, 362
 - map2, 346, 363
 - MemConfig, 177
 - MemnotWrD0, 177
 - Memory
 - configuration
 - ASCII output, 180
 - customized, 177
 - file, 192
 - in PAL, 177
 - in ROM, 177, 200
 - PostScript output, 180
 - standard, 177, 185
 - table, 190
 - configurer, 177
 - command line, 178
 - default configuration, 180
 - errors, 191
 - input parameters, 182
 - interactive operation, 180
 - output files, 180
 - disassembly, 308
 - Hex display, 129
 - inspecting, 310
 - interface, configurable, T4 and T8 series, 177
 - mapper, 271
 - command line, 272
 - errors, 281
 - read cycle, 188
 - write cycle, 189
 - Memory dumper, 175
 - command line, 175
 - error messages, 176
 - Memory map, 134, 311
 - boot from link (network), 97
 - boot from link (single processor), 94
 - boot from ROM, 98
 - collector output, 93
 - configurer, 54
 - memory.configuration, 198
 - MemStart, 94
 - MemWait, 185, 189
 - connection error, 191
 - MODIFY**, 145
 - Module data, listing, 245
 - MONITOR**, 148
 - Monitor page
 - See also Debugging
 - commands, 119
 - default address, 119
 - display virtual links, 141
 - Enter post-mortem, 140
 - exit, 140
 - simulator, 305
 - Monitoring the error status, 320
 - Motorola S-record format, ieprom, 202
 - MS-DOS, 325, 416
- ## N
- Network, dump, 135
 - listing, 250
 - Next error, 127
 - Non-bootable files, format, 91
 - Non-configured programs.
 - See icollect
 - notMemRd, 184

notMemS0, 184
 notMemS4, 184
 notMemWrB, 184
 Numerical parameters,
 interpretation by *isim*, 306

O

Object code, displaying, 237
 Options
 specify transputer target, 339
 standard, 325
 unsupported, 326
 Out of memory errors,
 idebug, 166
 output.address, 200
 output.all, 199
 output.block, 199
 output.format, 199

P

PAGE DOWN, 123
PAGE UP, 123
 patch, 346, 364
 codefix, 365
 datafix, 366
 extoffset, 367
 limit, 368
 modnumber, 369
 staticfix, 370
 Path searching, 326
 Porting C, 8
 Pragmas
 See also #pragma
 icc, 15
 Preprocessor
 directives, 12
 use with assembler, 342
 Priority, 138

ProcClockOut, 184, 185
 Procedural interface data, listing,
 247
 Process
 memory map, 139
 queue, 312
 displaying, 138
 Processor
 names, 133
 types, 333
 Protocol, *iserver*, 383

Q

Queues
 process, 138, 312
 timer, 312
 Quit
 debugger, 138
 simulator, 311

R

R-mode programs, 108
 RAM, 92, 98
 Read, strobe, 184
REFRESH, 122, 142
 Refresh period, 184
 Registers
 assigning value, 312
 memory dump, 176
RELOCATE, 122, 141, 146
 Reset, 116
RESUME, 122, 142, 145
 Resume program
 from debugger, 132
 from simulator, 310
RETRACE, 122, 141, 146
 Right shift, 8
 ROM, 92, 98, 195

Root transputer
 debugging, 107
 loading over, 317
root.processor.type, 198
Run queues, displaying, 138, 312

S

Scalar workspace, 92
Scheduling lists.
 See Process queues;
 Run queues
Screen definitions, 414
Screen size, 414
SEARCH, 147
Search path
 #include, 14
 configurer, 54
 conventions, 326
 icc, 7
Select process, 136
Select source file, 127
Selective linking, 228
Selective loading, libraries, 210
Session manager, 284, 288
 configuration file, 286
setconf.inc, 53
Shift right, 8
Show debugging messages, 139
Signedness of **char**, 8
Simulator, 303
 batch command files, 313
 batch commands, 313
 batch mode, 313
 booting program, 311
 command
 definitions, 307–313
 summary, 307
 command line, 303
 commands, 306
 errors, 314

 options, 304
 starting a program, 309
size, 346, 371
Skip loader, 317
 command line, 318
 command line options, 318
 errors, 321
sourcefile, 346, 372
srecord. See **output.format**
Stack
 checking, 5, 16
 position in memory, 54, 86
stack.buffer, 89
Standard memory configuration,
 185
Standards, file extensions, 327
start.offset, 200
Static area, position in memory, 54,
 86
Static data, 87
 memory map, 9
Static variables, memory map, 271
Subsystem
 connecting, 116
 reset, 318
Symbol data, listing, 241
Symbolic debugging, 142

T

T-mode programs, 108
T4 series, configurable memory
 interface, 177
T8 series, configurable memory
 interface, 177
Target, transputer, 333
Target files, for **imakef**, 254
Target transputer, command line
 options, 339
TCOFF, listing files, 250
Text files, listing, 250

textname, 346, 373

Timer queues, displaying, 139, 312

Timing data, 186

Tm, 184

TOGGLE BREAK, 145

TOGGLE HEX, 147

toolname, 346, 374

Toolset

- documentation, xx
- conventions, xxi
- standards, 325

TOP, 122, 141, 146

TOP OF FILE, 123, 147

Traceback information,
in ROM, 202

TRAM, 317

trams.inc, 53

TRANSPUTER, 286, 287, 292

Transputer

- accessing, 284
 - on a remote host, 293
 - on the local host, 293
- inline code, 21
- simulator, 303
- targets, 6, 333
 - command line options, 339

Trigraphs, 24

U

UNIVERSAL, 7

Unix, 325, 417

Unresolved references, 228

Unsupported options, 326

Update registers, 139

User link, 284, 291

V

Vector space, 92

- position in memory, 54, 88

Virtual memory, 227

Virtual routing, disable, 52

VMS, 325, 416, 417

W

Wait

- connection, 185
- race, 185
 - error, 191
- states, 185

Warnings

- See also Error messages
- selective suppression, **icc**, 16

Waveform diagrams, 188

Wired down, 116

Wired subs, 116

word, 346, 375

Write

- mode, 185
- strobe, 184
- to memory, in **idebug**, 140

Z

z, command line option, 326